Project Draft/Milestone 3, Project 1

Introduction

Perhaps one of the most common tasks a human does day to day is to write something down. Humans intuitively look at a document and begin to process the words. The question at hand here is it possible to train a computer to do the same thing and to recognize hand written text? If a computer can be trained to do that, what other kinds of tasks can be performed with the knowledge gained? If those all can be done, how one uses these features within context and apply them to pipelines or into business applications? It is with those questions in mind, the research for this paper began. We will be covering the history of people trying to solve this problem. One of the other more interesting aspects of the research for this paper was also participating in a Kaggle competition, and we'll be covering the lessons learned from participating as well.

Computer Vision is the art and practice of creating a computational model which mimics the human eye. Computer vision takes aim at an engineering point of view and seeks to create autonomous systems which are able to replicate the tasks that the human visual systems performs. Computer vision is a field which is known as being notoriously difficult. There are almost no problems within the field that have completely and satisfactorily solved. A good example of this is facial recognition. The human brain is able to break down facial recognition problems despite variations such as lighting, view points, facial expressions, and many other factors (Huang, 1996).

Kaggle competition is a competition platform for data science. They host data sets and a wide variety of competition types to the data science community in order to challenge individuals regardless of where they are within their data science careers. Kaggle hosts several competition types, including featured which are what they are best known for, which feature a cash prize. The getting started competition type is for some of the less complicated and more approachable types of problems trying to be solved with data science. They are meant to attract new comers and offer a host of tutorials associated with them ("How to use Kaggle", n.d).

The Digit Recognizer competition falls into the getting started category. The competition uses the MNIST ("Modified National Institute of Standards and Technology") data set. It is known as the de facto "hello world" data set for computer vision. It was originally released in 1999. The data features 2 files, a test and train CSV. The data in the MNIST data set is a set of images of hand written numbers 0-9. Each image is a 28 by 28 pixel image, for a total of 784 pixels. Each pixel is a value between 0-255 based on the shading of the image. The goal of the competition is to use computer vision in order to classify the images of the MNIST dataset. The competition is scored by the number of correctly labeled images (KAGGLE).

Background

Neural Networks

Before digging into the subcategory of convolutional neural networks typically used for digit classification, it would be helpful to review neural networks as a whole. The neural network we will be examining is the perceptron. Perceptrons were first conceived in the 1950s by Cornell psychologist Frank Rosenblatt. A perceptron works by outputting a single binary result from several binary inputs. The work that Rosenblatt brought to perceptrons was the use of weights. Weights are real numbers that are used to determine the importance of the input to the output (Nielsen 3, 2015).

When perceptrons are bundled together, they can be used for a surprising variety of tasks. While extremely powerful, one of the biggest drawbacks is that a slight change to the weight can change the entirety of the output. For example, within the context of digit recognition, say for example the number 4 is classified correctly, but not the number 9. A slight change in the weights of the model now allows for the number 9 to be correctly classified, but now the perceptrons no longer classify the number 4 correctly. This problem was solved by creating a new type of neuron similar to the perceptron called the sigmoid neuron. The benefit of using a sigmoid neuron is that the weights are able to be more fine tuned. This allows for a change in weight and bias to result in a small change to the output. By limiting the impact to the output, the network is able to be fine tuned as it learns. This takes the form of the input not longer being binary, but a decimal point value somewhere between 0 and 1 (Nielsen, 8, 2015).

When we bundle either perceptrons or sigmoid neurons, in essence, we get what is generally referred to as a neural network. A neural network is typically comprised of 3 layers, an input layer, a so called hidden layer, and an output layer. The first layer is the input layer, and the neurons contained within this layer are called input neurons. The middle layer, known as the hidden layer can actually be comprised of multiple layers. The term 'hidden' is used because the inputs and outputs of these layers are not revealed. The last layer is the output layer, which are the ultimate results you are trying to achieve. Looking down to figure 4, we have 4 layers. The input layer feeds to 2 hidden layers, and a single neuron is the output layer. This type of network is often also referred to as a multilayered perceptron (Nielsen, 11, 2015).

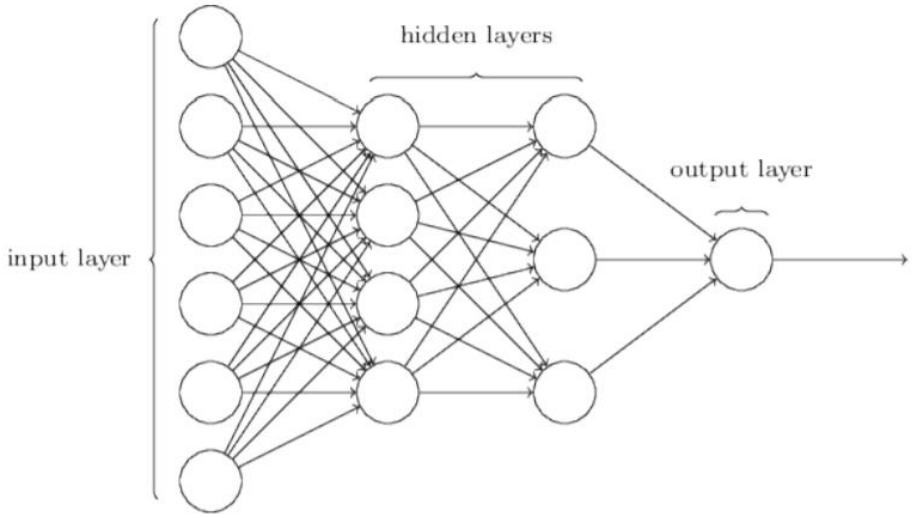


Figure 1: Nielsen, 2015

The techniques and image above help describe the original network types, and they're a sub class called feedforward neural networks. In this context, feedforward means that the data only goes in one direction. Data is fed to the input layer, and it only goes in one direction (Nielsen, 11).

When training a model, there needs to be a way in which feedback is passed in order for the model to adjust. Models are typically scored with what is known as the Mean Squared Error (MSE). The MSE is a measure of how closely a fitted line matches to its data points. For every value, you take the distance vertically and then square the value. Those values are then added up and averaged. The smaller this

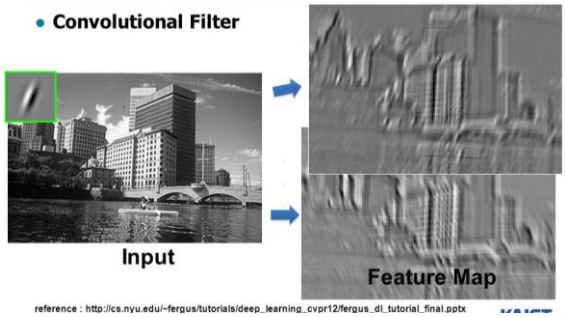
metric is, typically the more accurate your model is (What are Mean Squared Error and Root Mean Squared Error?, N.D).

As you train your model, in order to reduce the MSE, an algorithm known as gradient descent is typically used. Gradient descent is a heavily used optimization algorithm. The base version of gradient descent, known as vanilla or sometimes batch gradient descent, computes the gradient in relation to the entire training dataset. Batch gradient descent can be slow moving and can be not a great fit for a dataset that does not fit entirely into memory. Another major drawback to batch gradient descent is that it does not allow for updates to an online model. Another form of gradient descent is known as Stochastic gradient descent (SGD). SGD performs parameter updates for each of the training examples. By performing each update one at a time, it is much faster than batch and can be used to update the weights in online models without having to retrain. The last major type of gradient descent is the mini batch. This takes features from both worlds and makes an update for every *n* number of examples. With this method, the variance of the parameter updates is reduced, which ultimately can lead to more stable convergence. The range of batch sizes typically range from about 50 to 256, but that can certainly vary depending on the application. Mini batch is generally the algorithm used when training neural networks. It has become common enough that it has become referred to as SGD, despite the difference between mini batch and pure SGD (Ruder, 2016).

The technique of back propagation is the short hand term for backward propagation of errors. It is a technique which uses the gradient of descent for weighting the different variables. The process of back propagation is essentially a learning algorithm. Within a neural network, the process repeatedly adjusts the weights that the network uses to make predictions. The goal is to minimize the difference between the actual output and the desired output. This technique was first described in David Rumelhart's landmark paper in 1986 (Rumelhart, 1988).

The last background topic to be discussed is convolutions. Convolutions are perhaps one of the most heavily important concepts within deep learning right now. At its core, a convolution is defined as "an integral that expresses the amount of overlap of one function g as it is shifted over another function f," (Weisstein, N.D.). In essence, what this means is that a convolution is a mathematical method of blending together one mathematical function with another. Outside of the context of image classification, convolutions are heavily used within physics and engineering to simplify complex equations (Dettmers, 2015).

To understand how convolutions can be applied to images, it is first important to break down how a computer will see an image. Images can be represented by a matrix of data. Take for example a black and white image 28 by 28 pixels in height and width. This leads to a total of 784 pixels or values within the matrix. Each value is scored from 0 to 255 to represent its intensity, with 0 being completely white and 255 being completely black. If an image is color, then in essence, a layer to the matrix is added which represent the respective color, such as blue, red, green. Computers will observe images as a matrix of 28x28x3 for example. In a convolution, again we are taking the combination of two things, so on the one hand, you have this matrix of image definition. On the other hand, is a single matrix known as a convolution kernel. The output of an image definition and a convolution kernel is an altered image, and one is generated for each color channel for the image. The output of this is often referred to as a feature map (Dettmers, 2015).



There are dozens of types of feature maps available. Feature maps can produce many derivative images which make it easier for a network to learn off of. This can include sharpening the image, blurring the image, and many more. This process of transforming the raw data, and feeding it into the model is known as feature engineering. Within the context of neural networks which use convolutions, these are known as convolutional neural networks (CNNs). Convolutional networks get better as we train them to filter a particular feature map for the relevant information to make a prediction. CNNs do this process automatically, and this is known as feature learning (Dettmers, 2015).

Data Overview

The data used for this project was both the data set itself and derivative data of the Modified National Institute of Standards and Technology database (MNIST) data set. The MNIST data set is very commonly used for training models. The data set is a set of 60,000 training images and 10,000 testing images. The images are black and white images, normalized into a 28x28 box of hand written digits (LeCun, N.D.). The data is provided in several formats, and the one that was used here is contained within a CSV file. On each row of the file, there are a total of 785 columns, 784 columns to represent the pixels within the 28x28 image, and 1 column to represent the correct label of the data.

An additional subset of this same data was published onto Kaggle's website. The addition subset of images used is a random sampling of images from the MNIST data set. The difference is that instead of a CSV, the files are stored in JPEG format. The added benefit of this is that the images are in a format which is more likely to be used in machine learning problems. The data was contained within a Zip File within folders which indicated the correct label.

The last data set used was an additional set of 10 images, one for each of the 10 digits for 0-9. The data set was generated using Google Jamboard running on an Ipad with an Apple Pencil. Each digit was then saved individually into a separate JPEG file for each respective number. The images were then stored within a Zip file, and were labeled by filename.

Software

The software packages used for this project included the Kaggle Kernel as well Jupyter Notebook. The Kaggle Kernel is a robust hosted version of the popular Python Notebook environment Jupyter Notebook. The project was first run on a Kaggle Kernel before transferring to a local Jupyter Notebook environment. The solution was written using Python, with several imported packages. Those imported packages include Numpy, Pandas, OS, SKLearn, TensorFlow/Keras, Itertools, and IPython.display.

Data Preprocessing

All of the data was processed into a datafame. The dataframe had 3 columns, the label, the file source, and a Numpy Array of the image vectors, 28 x 28. Depending on the data source, the preprocessing steps varied on the data. For the standard MNIST data, the data is contained within a CSV sheet. The data was then read directly into the dataframe, and list comprehension was used to organize into the 3 columns. The 784 columns of the columns were normalized into a 28 x 28 Numpy Array. The JPEG formatted data were preprocessed using list comprehension. The labels were extracted from the filepaths and read into the data frame. The generated data was loaded into a list with the file paths. List comprehension was used to assign labels.

For the images stored in JPEG format, a function was created to move them into a Numpy Array of 28 x 28. The array took 1 parameter, the image filepath. The function used Keras preprocessing function load_img, img_to_array and returned a Numpy array. Using the .apply method from the Pandas dataframe, the function was applied to the filepaths contained in the data frame. The end result was the 3 column dataframe, one for the label, one for the filepath, and one which contained the normalized Numpy array for the image. Following this, the data was then split using SKLearns train_test_split function. The data was split into test and train subsets. The test size was 40%, with a random state of 101.

Model Creation

The model created was a Keras Sequential Model. A Keras Sequential model is a model that is created by passing a list of layers you wish to add to the network. The first layer of the model was a 2D convolutional layer. The second layer is a Flattening Layer, followed by a dropout layer. The output layer is a Dense Softmax layer. The output layer has 10 potential outputs, which represent digits 0-9.

Results

The results of this analysis come in 3 forms. The first form is empirical results per the rules of the Kaggle competition. The second is the additional data set, and finally is the generated data set. For the results of the Kaggle competition, the end RMSE score was 0.96285. This left my submission as a tie with a user that created a CNN using PyTorch. This result was tied at position 2146 on the leaderboard, out of the total 2585 participants in the competition. In the figure below, the leaderboard scores are visualized using a simple line graph. Scores for the competition are highly clustered together, with the exception being the extreme outliers.

Figure 2 Kaggle Leaderboard Scores Visualized

When the JPEG versions of the images were run through the model, they were scored with a RMSE of 1, meaning that they had full accuracy. The generated data set also achieved a RMSE score of 1, resulting in complete accuracy.

Results Analysis

In terms of results the meaning we are able to extract from the scores varies by the data set. The main results which have the most weight behind them are the results of the Kaggle competition. The competition used a large and statistically significant enough data set in order to score the models, an additional 28,000 records. The JPEG data set was a subset of the training data provided, and had approximately 600 images. Those images were within the data set that was used as part of the training data for the model. In essence, the model had already "seen" those examples. Due to that fact, it is neither surprising nor does it lead to any additional questions due to the fact that it had 100% accuracy. While the scoring was not surprising, the use of the data set achieved the goal of providing additional context for pipeline and other preprocessing preparations that are typically required in other contexts. The last part of the analysis that was conducted was the generated data set. The generated data set results came as an interesting surprise as well. Given the size of the data set, 10 handwritten images, and the image clean up and preprocessing done before going through the preprocessing, it is difficult to draw conclusions from the data. Had this data set had more samples, more conclusions might be able to be derived from the results. Each data set represented a different stage of potential input data a model might receive. The pure MNIST data represented an ideal data set, with lots of samples, and preprocessed in a format which was almost ready for machine learning. The JPEG derivative is an example of data that had already been seen, but required additional preprocessing. The generated data set represented an ideal solution for new data that was never before seen. Additional exploration with a larger sample size of the generated data would be a good follow up study.

Conclusions

The outcome of this analysis definitely proves the ability for computers to generally be able to perform this task just as close to a human. There's definitely a number of other things to bring up about this and what this means in the greater context in order to put this into production. While there are not great conclusions to be drawn from how well the model scored the data set generated, it brings up a lot of lessons learned. The generated data was more or less generated in an unusable format. I had to separate the characters, and adjust the color scale manually. In the context of recognition, very few times will the data be neatly separated out in advance. Rarely, do you know all the potential values which you might be getting. In the use case of postal zip code recognition, you have high certainty that the zip code will all be digits. In realistic situations, how the data is generated is also going to be of concern. What image format was the image generated in? How are you going to separate the individual digits from each other so each one can be individually identified? What if you are looking at a corpus that includes letters? Breaking down the problems, respectively, you need to account for image variety and to have software normalize the images and colors. Separate software to identify where the individual digits are for individual recognition. Additionally, natural language processing would be in order, to help correctly classify so that within context. For example, within context would mean that the letter O in the middle of word is not classified as a zero.

Future Studies

One of the limitations of this analysis was that the data was in greyscale. A future study would perhaps examine more colors beyond just the grey images used in the training data set. The data would again need to be formatted in a uniform single digit per image training session. This would test a CNN networks ability to adjust to

References

Dettmers, T. (2015, March 26). Understanding Convolution in Deep Learning. Retrieved from http://timdettmers.com/2015/03/26/convolution-deep-learning/

Digit Recognizer. (n.d.). Retrieved March 27, 2019, from https://www.kaggle.com/c/digit-recognizer

How to use Kaggle. (n.d.). Retrieved March 27, 2019, from https://www.kaggle.com/docs/competitions

Huang, T. (1996). Computer vision: Evolution and promise.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation*, *1*(4), 541-551.

LeCun, Y., Cortes, C., & Burges, C. J. (n.d.). THE MNIST DATABASE. Retrieved from http://yann.lecun.com/exdb/mnist/

Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Determination Press.

Ruder, S. (2016, January 19). An overview of gradient descent optimization algorithms. Retrieved from http://ruder.io/optimizing-gradient-descent/index.html#minibatchgradientdescent

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1988). Learning representations by back-propagating errors. Cognitive modeling, 5(3), 1.

Weisstein, Eric W. "Convolution." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/Convolution.html

What are Mean Squared Error and Root Mean Squared Error? (n.d.). Retrieved from https://www.vernier.com/til/1014/