Improving the Accuracy of Neural Networks

Emily Nguyen

September 4, 2018

Abstract

This paper aims to answer the question: what are effective ways to improve a neural network's accuracy? A neural network was created to classify individuals' races according to photos of their faces. This neural network's performance was recorded after changing its parameters to test how its accuracy is affected. This project also compares the results of normal neural networks vs convolutional neural networks (CNNs) and using black and white images vs RGB images. The results indicate that networks tend to perform better when there are more epochs, more layers, and/or more training data. There appears to be neither much of a difference in accuracy between normal neural nets and CNNs nor between black and white and RGB images. The lack of improvement is most likely because the dataset is not very diverse. However, this study did not cover all possible changes in parameter, so future studies should attempt more possible permutations.

1 Introduction

Neural networks are a form of machine learning that have a wide range of applications, such as image recognition, audio recognition, and text processing. An example of an application for neural networks is identifying individuals passing through customs at the airport using face recognition. In this example, it is important for the network to be accurate to avoid incorrectly identifying citizens and permanent residents. On a much larger scope, a higher accuracy percentage is generally desired, which necessitates research concerning how to improve a network's accuracy.

This paper begins with a brief background on neural networks and convolutional neural networks. This explanation is followed by an introduction to the problem, a recount of the methods taken to attempt to address the problem, and a discussion of the results and future directions.

1.1 Neural Networks

Feedforward neural networks have layers of neurons that take in inputs, scale the inputs with their own weights (W) and biases (b), and output a real number ($f(Wa+b)$) to the neurons in the next layer.

The output of each neuron is calculated by applying a nonlinear activation function (f) to the inputs from the previous layer. A few popular nonlinear activation functions include: the sigmoid function, the hyperbolic tangent function, and a rectified linear unit (ReLU). Nonlinearity is preferred for a network because data is usually nonlinear, so a nonlinear neural network would better approximate the results than a linear one. Additionally, using a linear activation function would essentially make the neural network only one layer deep despite the number of layers in the set up.

A **dense** (or fully connected) network is a network in which the neurons in each layer are connected to all of the neurons in the next layer. For this type of network, the inputs for each layer are

$$a^{[1]} = x \tag{1}$$

$$a^{[l]} = f(W^{[l]}a^{[l-1]} + b^{[l]}) \; for \; l = 2, 3, ..., L \tag{2}$$

where x is the input data and L is the total number of layers in the neural network [HH18]. Not all neural networks are dense, but the one used in this study is. The output of the final layer of the neural network is

$$F(x) = f(W^{[L]}f(W^{[L-1]}...f(W^{[2]}x + b^{[2]}) + ... + b^{[L-1]}) + b^{[L]}]) \tag{3}$$

which is used to help train the neural network [HH18].

1.2 Training

Neural networks are trained by evaluating and minimizing their cost function

$$Cost(W^{[2]}, ..., W^{[L]}, b^{[2]}, ..., b^{[L]}) = \frac{1}{2N}\sum_{i=1}^{N} ||y(x^{\{i\}}) - F(x^{\{i\}})||_2^2 \tag{4}$$

where N is the number of data and $y(x^{\{i\}})$ is the expected output. The data will have an expected output if it is labeled [HH18]. This means that the training will be supervised since the network has the correct answer to compare to. Training is necessary because normally, a network's weights and biases are initialized randomly, so the network's first guesses are expected to be very inaccurate. Since the cost function depends on the weights and biases, these parameters need to be updated to decrease the cost.

The cost takes the sum of the quadratic norms of the difference between the expected outputs and the network's outputs and divides by the total number of data points to calculate an average cost (or loss). Note that the cost is a function of the weights and biases, so minimizing the cost function would require updating the weights and biases. Normally, finding the minimum of a function means finding the critical points of the first derivative, but this is difficult to do for higher dimensional data. As such, there are optimization methods meant to help, like stochastic gradient descent, which is used in this project.

Gradient descent reveals the direction and magnitude for the weights and biases to change in order to minimize the cost function. This method requires taking the negative of the gradient of the cost function, multiplying the gradient by a learning rate (η), and adding this product to the current weights and biases. This looks like

$$p \to p - \eta \nabla Cost(p) \tag{5}$$

where p is a vector of the weights and biases. This process is repeated until p can approximate well the vector that would minimize the cost. However, gradient descent is expensive to compute since it requires computing the gradient of every data point at every iteration, so another method is introduced: **stochastic gradient descent** (SGD). SGD differs from gradient descent in that it chooses one training point at random and uses that training point's gradient in place of the mean of the gradients of all of the training points. The path to finding the minimum becomes staggered in comparison to the path found by using gradient descent, but there are significantly fewer computations required [HH18].

To find the partial derivatives of the cost function with respect to each of the network's weights and biases, **back propagation** is used. Back propagation calculates the network's output in a forward pass and uses this output to work backwards through the network to find the partial derivatives. To simplify the equations for the algorithm, we introduce two variables: $z_j^{[l]}$, the weighted input for neuron j in layer l such that

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]} \; for \; l = 2, 3, ..., L \tag{6}$$

$$a^{[l]} = f(z^{[l]}) \tag{7}$$

and $\delta_j^{[l]}$, the error of neuron j in layer l such that

$$\delta_j^{[l]} = \frac{\partial Cost}{\partial z_j^{[l]}} \; for \; 1 \leq j \leq n_l, \; 2 \leq l \leq L \tag{8}$$

where $n_{[l]}$ is the number of neurons in layer l. Note that $\delta_j^{[l]}$ measures the sensitivity of the cost function with respect to each neuron's weighted inputs, so the cost function is minimized when all of the partial derivatives equal zero.

Back propagation takes the network's output (a^L) and uses it to find the error of the last layer, then the errors of the previous layers, and finally the sensitivity of the cost function with respect to the individual weights and biases. The algorithm follows as

$$\delta^{[L]} = f'(z^{[L]}) \circ (a^L - y) \tag{9}$$

$$\delta^{[l]} = f'(z^{[l]}) \circ (W^{[l+1]})^T \delta^{[l+1]} \; for \; l = 2, 3, ..., L-1 \tag{10}$$

$$\frac{\partial Cost}{\partial b_j^{[l]}} = \delta_j^{[l]} \; for \; l = 2, 3, ..., L \tag{11}$$

$$\frac{\partial Cost}{\partial W_{jk}^{[l]}} = \delta_j^{[l]} a_k^{[l-1]} \; for \; l = 2, 3, ..., L \tag{12}$$

where the operation $\circ$ is the Hadamard (or component-wise) product of two vectors [HH18].

In simpler terms, the final output of the neural network helps calculate the cost. Therefore, the partial derivative (sensitivity) of the cost can be calculated with respect to the output of the network. Since the network's output depends on the weights and biases from the previous layer, the sensitivity of the cost with respect to weighted inputs in each layer can be calculated. Finally, since these inputs depend on the weights and biases in the network, the partial derivative of the cost with respect to each weight and bias can be calculated. In essence, this is the chain rule. For intuition: changing a weight or bias changes the input for the next layer in the neural network, which changes the the next layer and so on until the output of the network is changed, which changes the cost of the network.

These partial derivatives calculated by back propagation are then used to update the weights and biases with stochastic gradient descent.

1.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) filter the data and reduce the input data by using convolutions and kernels. As such, CNNs are useful for identifying key features of images and dealing with large inputs [HH18].

CNNs apply a filter (or a kernel), which contains weights, through the data to attempt to discern important features. As the network trains, the weights in the kernel are updated to better detect significant parts of the images, like edges or in the case of this project, eyes and noses. The output of a convolutional layer is multiple feature maps of the extracted features. Since this creates a bigger input for the next layer than the original image, another technique used for reducing the data size is pooling. This project uses max pooling, which goes through the data and for every 2 x 2 block in the input tensor, takes the maximum value and discards the other three values. The max values are all put into another matrix that is smaller than the input matrix. Pooling preserves the important parts of each feature while getting rid of unnecessary parts. An additional transformation the images normally go through after pooling is being passed through with a ReLU. A ReLU goes through the images and replaces any negative values with a zero [OAHY14].

A network can have multiple convolutional, pooling, and ReLU layers to pick up more specific features or to further reduce the amount of data. After all of these layers, the last output images are put into a fully connected layer, and the values in these tensors help determine the output of the entire network.

2 The Problem

Neural networks help solve a myriad of problems that require image recognition and classification, such as recognizing a country's citizens and permanent residents going through customs at an airport. Due to the importance of a network's output in these types of applications, improving a network's accuracy is crucial to reducing errors that could have repercussions, like incorrectly identifying citizens and not letting them through customs quickly. This paper experiments with ways to increase a network's accuracy.

2.1 Approaches

A normal feedforward neural network and a CNN were created with the purpose of classifying individuals' races from images of their faces. The accuracy of each network was recorded after changes to the input or parameters, but the CNN maintained two convolutional and two pooling layers for all results.

Theoretically, the following changes would increase the accuracy of the networks: using RGB images instead of black and white images, increasing the number of layers in the network, increasing the number of **epochs** (which are training cycles where all of the training data passes through the network and the weights are updated once [HH18]), and moving some of the testing data into the training data (in most of this project's test cases, out of 2193 images, there are 1900 training images and 493 testing images [22.4% of the total data] but for this change, 200 images are moved to the training data to lower the percentage of testing images to 13.4%) , distorting the data (by flipping the images horizontally and vertically and saving these as new images in the data set to increase the size of the training data; the images might not look vastly different to humans but they are completely different images to the networks).

Batch size and learning rate are still debated parameters in that there is no established rule for the best batch size or learning rate to maximize the network's accuracy. Therefore, this paper attempts various batch sizes and learning rates to determine the best values for this specific project. The RGB images were passed through the convolutional neural network in two different ways. The first way is by taking the red, blue, and green images of each picture and concatenating all three into a single matrix. The other way is to use a three-dimensional filter to go through all three channels at once in the convolution layers. Note that for the CNN, unless otherwise stated in the method, black and white images were used, and the "normal" method is a control for the other methods in Figure 1 (increasing the amount of training data, distorting the images, and using RGB images). This means that for the "normal" CNN method, there are 493 testing and 1700 training images, and the images are black and white and not distorted.

The python code for the convolutional neural network can be found on the tensorflow website, and the code for the normal neural network can be found on the Neural Networks and Deep Learning[1] website. The images used for this project are from the Wilma Bainbridge's website [WABO13].

% Test Data/ Total Data	Epochs	Number of Layers	Sizes of Layers	Batch Size	Learning Rate	Accuracy
22.4%	100	4	130*200, 100, 50, 7	10	3.0	80.93%
22.4%	100	4	64*100, 100, 50, 7	5	3.0	83.16%
22.4%	100	4	64*100, 100, 50, 7	10	1.0	84.58%
13.4%	100	3	64*100, 100, 7	10	3.0	86.69%
13.4%	300	4	130*200, 100, 50, 7	10	3.0	87.03%
13.4%	100	5	64*100, 100, 50, 50, 7	10	3.0	87.03%

Table 1: Results for black and white images using a normal neural network

2.2 Results and Discussion

The results in Table 1 and Table 2 indicate that normal neural networks tended to perform better when: there are more epochs, there are more layers, and/or more training data, which is expected. However, there are not conclusive results for whether decreasing batch size or learning rate improves the network. Overall, there does not appear to be a significant difference in performance between using gray and RGB images for the normal neural network. Comparing the results in both tables with the ones in Figure 1 indicates that normal neural networks appear to perform only slightly better than CNNs, which is unexpected because the CNN should have outperformed the normal neural network.

Comparing the results for the CNN only, moving some testing data into the training data and distorting the images (which are supposed to increase the amount of data in the training set) worsened the network's performance while using RGB images slightly increased the CNN's

[1] http://neuralnetworksanddeeplearning.com/

% Test Data/ Total Data	Epochs	Number of Layers	Sizes of Layers	Batch Size	Learning Rate	Accuracy
22.4%	100	4	130*200*3, 100, 50, 7	10	1.0	3.4%
13.4%	100	4	130*200*3, 100, 50, 7	10	3.0	79.5%
22.4%	100	3	130*200*3, 100, 7	10	3.0	82.75%
22.4%	100	4	130*200*3, 100, 50, 7	5	3.0	83.98%
22.4%	200	4	130*200*3, 100, 50, 7	10	3.0	84.3%
13.4%	300	5	130*200*3, 100, 50, 50, 7	10	3.0	85.67%

Table 2: Results for RGB images using a normal neural network

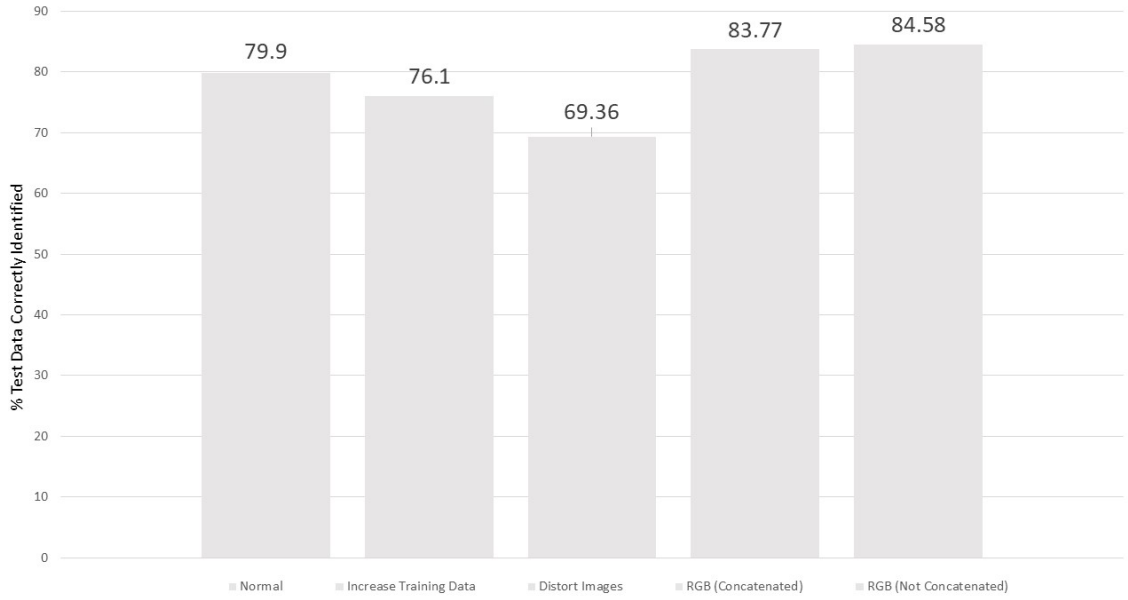

Figure 1: Results for RGB and gray images using a CNN

accuracy. Not concatenating the RGB images led to only a small improvement over concatenating them.

3 Conclusion and Future Work

It makes sense that increasing the number of epochs, layers, or amount of training data improves the network's performance because these changes provide the network with more practice. However, the lack of improvement of using a CNN over a normal neural network and also of distorting the images is most likely because the dataset is not diverse. There is a significantly higher proportion of images of Caucasian individuals than individuals of other races. As such, the network does not have sufficient data to confidently identify other races, and the CNN cannot pick up on features that are significant enough. This is also the case for the negligible difference between using black and white and RGB images.

In terms of future directions, this project should be performed again with a more diverse dataset to provide the network with adequate practice for identifying all races instead of (most likely) being reduced to identifying the binary of Caucasian images and not Caucasian images. A good first step could be to break down the network's errors and see which races are more often

incorrectly identified and how they are identified. Additionally, each parameter change should be tested multiple times to ensure precision and to get more conclusive results for how changing the batch size and learning rate affects the network. Repeated trials would also make the results more robust to outliers, like the first result in Table 2 that has an accuracy of 3.4% (which most likely happened because the network either stayed at a local minimum instead of the absolute minimum or a better local minimum or because the network couldn't update quickly enough with the smaller learning rate).

References

[HH18] Catherine F. Higham and Desmond J. Higham. Deep learning: An introduction for applied mathematicians. *arXiv preprint arXiv:1801.05894*, 2018.

[OAHY14] Hui Jiang Li Deng Gerald Penn Ossama Abdel-Hamid, Abdel-Rahman Mohamed and Dong Yu. Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP)*, 22(10):1533–1545, 2014.

[WABO13] Phillip Isola Wilma A. Bainbridge and Aude Oliva. The intrinsic memorability of face photographs. *Journal of Experimental Psychology: General*, 142(4):1323–1334, 2013.