**Image Recognition Using Machine Learning**

Elijah H Gallagher

Harrisburg University of Science and Technology

Project 1

Prof. Brian Grey

Aug 20th 2022

In my project I covered various algorithms to perform image recognition on the MNIST dataset and the CIFAR-10 dataset. After testing I found that the strongest performing traditional algorithm for the MNIST dataset was a support vector machine, whereas for the CIFAR-10 dataset a convolutional neural network was far superior to any other algorithm. In this paper I will go into detail about my process in creating these algorithms, and the respective challenges in doing so.

1. **Explaining MNIST data**

The MNIST data is a dataset of handwritten digits created primarily for educational purposes by LeCun et al., in 1998. It includes 60,000 training images and 10,000 testing images (see Figures A1 & A2 for an example). This data is formed in a 28x28 array with the value at (x, i) being an int between 0-15 to denote how bright the pixel is. I also drew ten handwritten digits to add to the testing set.

1. **Simple models on MINST data**

For the first iteration of my project I used available libraries to create my algorithms, primarily sklearn which contains many different algorithms. For this Project I used a Support Vector Machine, a Gaussian Naïve Bayes, a Decision Tree, a Random Forest, a K Nearest Neighbors, and a Gradient Descent algorithm.

Gaussian Naïve Bayes is a supervised learning method that uses the Bayes algorithm. It is effectively guessing the correct tag based on the frequency of each pixel in each image. For example, if the bottom left corner is only bright when we have a one, it will assume if that pixel is light that the image is a 1.

A decision tree is a discrete mathematics concept that can easily be applied to classification problems. There are two kinds of nodes, ones that make a binary decision (left or right), and leaf nodes which are functionally output.

A random forest model is just the sum of a bunch of decision tree models, generally between 7-13 trees where you take the majority prediction.

K Nearest Neighbors, generally called KNN, is a supervised learning algorithm which tries to plot the data points and guess what grouping the new sample belongs to.

Gradient descent is a stepwise approach which minimizes a cost function. It is used in both support vector machines as well as any kind of regression.

Support Vector Machines (SVMs) are supervised learning algorithms that do regression analysis. They can perform both linear classifications and non-linear classifications by approximating a higher dimensional relationship using a kernel.

Out of these algorithms, as seen in Table A1, an SVM outperformed the other algorithms in accuracy on the MNIST data while keeping training and classifying time low. Because it seemed to be the best performing algorithm, I explored the options that an SVM could provide along with trying to gain an understanding of the math behind them.

1. **Deeper look at Support Vector Machines**

SVMs are unique among classifiers I tested because they perform what’s known as a ‘kernel trick’ to approximate higher dimensional relationships. There are two types of kernels in SVMs, either a polynomial kernel, which is a n-degree polynomial where it is attempting to approximate an n-dimensional relationship, or an RBF kernel (radial kernel) which uses an infinite series to approximate the relationship between data points. The output of an SVM is very similar to linear regression, by drawing a single line between the mean of the data and then ‘bending’ that line at different points. If, after doing this, the SVM still cannot find a clear separation between the data, it will go up a dimension, meaning the SVM will add a term to the polynomial or infinite series.

1. **Overview of Neural Networks**

A neural network is just a way to explain an algorithm that is using nodes which are arranged into many layers. These nodes/layers are connected using weighted connections between them and are updated using an algorithm such as backpropagation to update the weights. For my project I used two types of neural networks (NN), a basic NN that is a forward-flowing model and a convolutional neural network (CNN). A basic NN is a forward-flowing model consisting of dense layers and contains at least two hidden layers. This is different than a convolutional NN, which is mostly a forward-flowing model consisting of convolutional, pooling, and dense layers, containing at least four hidden layers, and performs convolutions. These convolutions are a transformation that occur every time data is passed through a convolutional layer.

1. **Example of convolution**

A convolution in a CNN is not a mathematical convolution, it is repeated matrix multiplication to distort an image in some way (see Figure A3). In this figure, we are blurring an image by: selecting a matrix that is n by n where n is odd, padding the initial image with zeros so that our dot product does not have null values, and then performing repeated matrix multiplication where the ‘middle’ of our matrix is aligned over every pixel of the initial image to produce the next image. When it comes to image recognition blurring an image is not helpful. Almost all CNNs will use what are generally referred to as ‘edge detection matrices’ (see Figure A4). These will populate a layer of the CNN and allow it to recognize different groupings of edges to predict a tag.

1. **The NN models on MINST data**

The test results from the NNs and CNNs on the MINST data, shown in Table A2, show that they all have an accuracy within 0.39% of each other. This is important to note because the training time is not close to equal. We can see a jump in training time amongst the NNs that, while not important for our implementation, could be detrimental if our model was significantly bigger. Of note is that the CNNs take significantly longer to train and do not perform better. This is because our data is relatively simple and contains no color.

1. **Direction and preliminary conclusions**

Overall, the data it is not clear that an increase in layers had any positive impact on the accuracy of the model, and in some cases may even lower the model’s accuracy while increasing training time. This is compounded by the fact that our standard models like SVMs are performing almost equivalently to the NNs with even less training and classifying time. Because the CNNs did not perform better than the NNs, I decided to transition to using the CIFAR-10 dataset which I will fully explain in section 8 below. I suspect the CNNs will perform better on this type of data when compared to the NNs. This is mostly because in the MINST dataset we have greyscale images which can be represented as a 28x28 array, whereas colored images like in the CIFAR-10 dataset are represented by a 32x32x3 array.

1. **CIFAR-10 dataset**

The CIFAR-10 dataset, which was created by Alex Krizhevsky in 2009, provides a large colored image dataset, “The labeled subset we collected consists of ten classes of objects with 6000 images in each class. The classes are airplane, automobile (but not truck or pickup truck), bird, cat, deer, dog, frog, horse, ship, and truck (but not pickup truck).” (Krizhevsky, 2009, p. 34). Overall, this dataset provides a significantly more complex set compared to the MNIST because it is 32x32x3. The CNNs should also be significantly better than the basic NNs on this dataset because of their ability to detect lines and handle more complex data.

1. **Creating my own data to test**

For this project I want to compile a small sample of my own downscaled images to provide an extra set of testing data that I can use to validate my models. I downloaded ten images for each category to use as this extra set and included in my code a quick program to downscale images to 32x32x3, and to expand from a greyscale 32x32x1 to black and white 32x32x3 as necessary. One large problem with downscaling images is that you can lose information if the image is not scaled to its native resolution. For the CIFAR-10 dataset, all of the images are square images although not all images in my validation set are. This leads to a worse-than-expected performance on my validation images. However, after understanding how the downscaling can lose information, I think the difference is explainable by the resolution issue.

1. **Conceptual design for code**

I want my code to perform three main functions: load in and downscale images, train and save the models, and classify them. The models will be built in TensorFlow, made by Abadi et al., in 2016. It is a framework for building both the NN and the CNN I will be using.

1. **Implementation**

To start my implementation, I created a Python file (trainModels.py) to create the NNs and CNNs. trainModels.py is the simplest part of the implementation to create because once I understood how to put together the layers of a NN/CNN, it just became rather repetitive coding. Using TensorFlow trainModels.py creates four models and saves them so that they are not required to train every time you want to classify new images. This is important because training all of the models can take a lot of time, generally 10-15 minutes each. trainModels.py creates two CNNs and two NNs. One CNN and one NN will be intentionally overfitted (overfitting is a process of training the model so much that it is only reliable on the train/test dataset), by being bloated with extra layers and extra epochs (an epoch is an additional time the model is trained). This allows a user to see how the models with more training do not necessarily perform better than ones with less training.

The second part of the implementation was creating reshape.py. This was the hardest Python file to code because it required a large number of things that I had no experience in prior to coding this. Primarily, reshape.py needs to be able to take an image file and hand a downscaled version to the main program. For the resizing, I used a module cv/cv2 created by Bradski, (2000). Additionally, reshape.py needs to be able to read all the image files in a directory and send them all over to the main program at the same time. This was mostly done to save time at execution and make it easier to classify groups of images.

The primary function in reshape.py is reshapeAllImg which takes an optional parameter as a path (the folder containing the image files). First, this function will call a load\_all\_images function, that will load all of the images into an array and transform greyscale images into black and white images. This will give us a three-dimensional array which is necessary since all of our images will need to be in the same format for classification. Then this transformation is done again using cv/cv2. Once load\_all\_images is complete, it returns a list of all the image files in array form. Next, reshapeAllImg will resize all of the images to be 32x32 to match the CIFAR-10 dataset. At the end, this single function call will return an array with a tuple of the file name and an image of size 32x32x3 for every image in the directory. I did this for my personal testing since for my own data I was using the image names as the tags for the image. Given this, having the name in a tuple along with the image data was important for classification.

The last part of the implementation is classify.py. This is the file that is going to be the main program and the primary one I expect a user to interface with. This file is responsible for first attempting to load a copy of each saved model. If more than two of the models fail to load, it will call trainModels.py to retrain and resave a new version of these models. This process is rather slow and should only be used if the user is unsure that all four models are saved. Then classify.py will call reshape.py for the list of resized images, load and run them through the models, print the output to console (see Figure A5), and finally output a csv file. This csv is simply a list of all images and the tag the model gave them.

1. **Observations & Analysis**

I performed some testing to compare the performance of the CNNs and the NNs. Unfortunately, the NNs failed to adequately classify my validation images, only being able to correctly classify the images less than 50% of the time. On the other hand, the CNNs currently classified the image over 70% of the time. Therefore, I decided that the CNNs were not comparable to the NNs in this problem. I think this is because adding a third dimension to the images allows the CNNs to flex their ability to perform edge detection. Although both performed with over 90% accuracy on the MNIST dataset, the NNs’ performance drops off drastically because of the simplicity with which they classify images.

1. **Conclusion**

Overall, I have applied multiple algorithms to solve both a ‘simple’ image classification problem and slightly more complex image classification problem. I am happy with the end result of my project, especially since I was able to go above and beyond the initial expectations I had of myself. At the start of this project, I expected to end on being able to classify handwritten digits, and had no real expectation to advance to the colored images of the CIFAR-10 dataset. However, after getting familiar with the concepts and core ideas of image recognition and convolution, I am quite happy with what I produced.

**References**

Abadi, Mart’in, Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., … others. (2016). Tensorflow: A system for large-scale machine learning. In *12th $USENIX$ Symposium on Operating Systems Design and Implementation ($OSDI$ 16)* (pp. 265–283).

Bradski, G. (2000). The OpenCV Library. *Dr. Dobb’s Journal of Software Tools*.

deeplizard. (2017, December 9). Convolutional Neural Networks (CNNs) explained [Video]. YouTube. <https://www.youtube.com/watch?v=YRhxdVk_sIs>

Krizhevsky, A. (2009). *Learning Multiple Layers of Features from Tiny Images*. https://www.cs.toronto.edu/. Retrieved July 31, 2022, from <https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf>

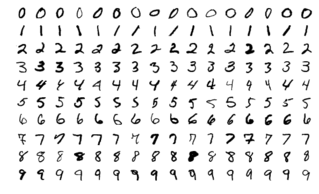
Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document

**Appendix A**

Tables & Figures

**Figure A1**

*Image of MNIST pictures*



**Figure A2**

*2D representation of a 0*

A picture containing text, crossword puzzle

Description automatically generated

**Figure A3**

*Example of convolutions matrix multiplication*

Graphical user interface, application, table, Excel

Description automatically generated

*Reprinted from deeplizard. (2017, December 9). Convolutional Neural Networks (CNNs) explained [Video]. YouTube.* [*https://www.youtube.com/watch?v=YRhxdVk\_sIs*](https://www.youtube.com/watch?v=YRhxdVk_sIs)

**Figure A4**

*Example of Edge Detection matrixes*

A screenshot of a game

Description automatically generated with low confidence

*Reprinted from deeplizard. (2017, December 9). Convolutional Neural Networks (CNNs) explained [Video]. YouTube.* [*https://www.youtube.com/watch?v=YRhxdVk\_sIs*](https://www.youtube.com/watch?v=YRhxdVk_sIs)

**Figure A5**

*Example of consol classifacation output*

Text

Description automatically generated

**Table A1**

*Tests using the basic algorithms on the MNIST dataset*

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Time to Train | Time to classify | Accuracy |
| Support Vector Machine | 0.111 Seconds | 0.106 Seconds | 96.88% |
| Gaussian Naive Bayes | 0.058 Seconds | 0.039 Seconds | 80.75% |
| Decision Tree | 0.127 Seconds | 0.036 Seconds | 74.97% |
| Random Forest | 0.142 Seconds | 0.128 Seconds | 75.31% |
| K Nearest Neighbours | 0.164 Seconds | 0.089 Seconds | 95.55% |
| Gradient Descent | 0.078 Seconds | 0.045 Seconds | 88.65% |

**Table A2**

*Test on the MNIST data using the neural networks*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Neural Network** | # of Layers\* | Time to Train | Time to Classify | Accuracy | Test on my data |
| Basic | 4 | 33.96 Seconds | 1.43 Seconds | 99.23% | 6/10 |
| Basic | 3 | 18.42 Seconds | 1.39 Seconds | 98.92% | 7/10 |
| Basic | 2 | 12.23 Seconds | 1.45 Seconds | 99.12% | 7/10 |
| Convolutional | 5 | 296.40 Seconds | 2.82 Seconds | 99.31% | NA |
| Convolutional | 4 | 245.66 Seconds | 2.44 Seconds | 99.13% | NA |