Assignment 2: Data Analytics Project

32513 Advanced Data Analytics Algorithms

Description of the business problem you are solving together with how this translates into a data-mining problem.

The team discussed a wide range of problems that could potentially be solved using data mining. The scenarios that caught our attention the most are handwritten exam papers, assignments, primary and high school homework, doctors dictation notes about patients condition in hospital and so on.

The common thread in the scenarios stated above is handwriting. In today's technology orientated era, there are still many fields that implement and use handwritten notes. The business problem here is classifying handwriting using machine learning. The intention behind this is to train the built model to analyze the images and predict the alphabet or digits they are. This is handwriting recognition. By solving this problem, handwritten records can be electronised in a much faster and orderly way and it would be easier to perform a word search on documents. "Post offices can use them to sort letters; banks can use them to read personal checks and so on" (Ciresan et al. 2010).

Data analytics is also being explored to solve this problem. Using data mining classification techniques like Neural Network and k-Nearest Neighbours. The classifier will learn to recognize each character in the training dataset and accurately predict the character of the test dataset. The classifier later uses this knowledge to classify handwritten characters in the test dataset.

For this project the team decided to explore two different data: MNIST (Mixed National Institute of Standards and Technology) handwritten digits dataset and NotMNIST dataset. NotMNIST dataset represents hieroglyphic fonts for letters between A to J (Flanagan 2015), which makes it much harder to recognizer a character or alphabet. Therefore, a combination of these two dataset classifying using the same model makes it an interesting classification problem because different settings would need to be explored to satisfy both data. The team has decided to use python to build our data mining models.

A description of your exploration of the dataset highlighting interesting or important things you found (roughly 5–10 pages with figures).

For this project the team decided to explore two different data: MNIST (Mixed National Institute of Standards and Technology) handwritten digits dataset and NotMNIST dataset. NotMNIST dataset represents hieroglyphic fonts for letters between A to J (Flanagan 2015), which makes it much harder to recognizer a character or alphabet. Therefore, a combination of these two dataset classifying using the same model makes it an interesting classification problem because different settings would need to be explored to satisfy both data.

We are using Yan LeCun, MNIST dataset (LeCun 1998). The MNIST is a database of handwritten digits between 0 to 9 and it is a subset of a larger set of NIST database (LeCun 1998). So there can only be 10 classes to classify a handwritten digit.

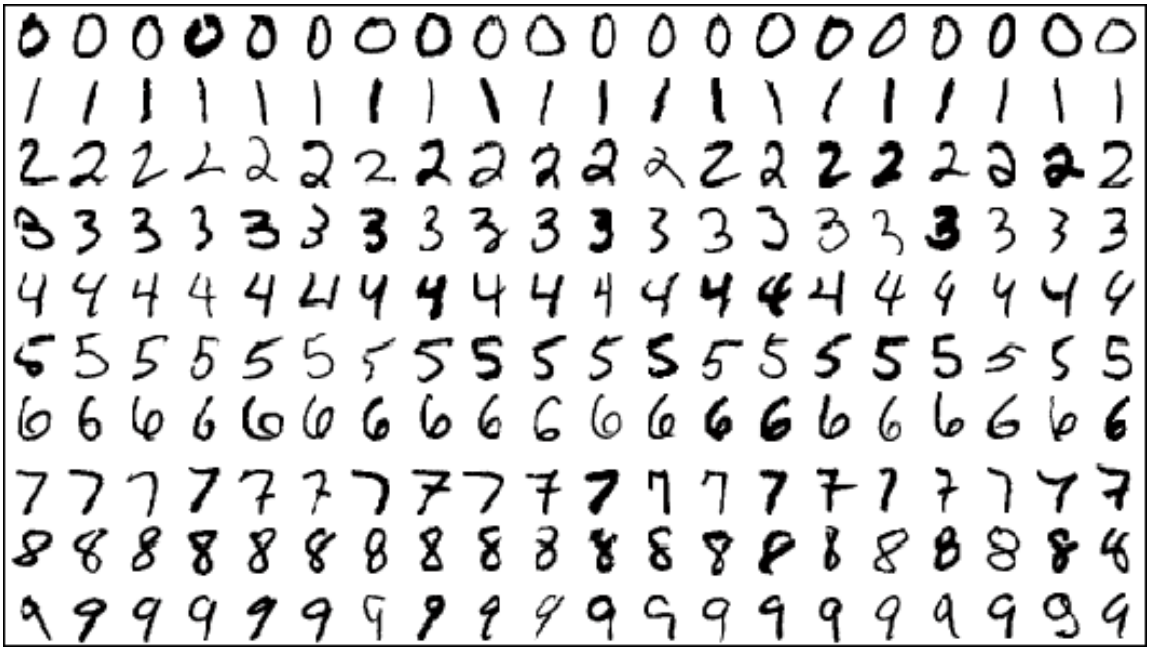


Figure above shows some of the handwritten digit images used in the MNIST dataset (Winder 2015). Each digit is drawn in a different way.

NIST has multiple special databases containing handwritten digits and special database 3 was originally used as their machine learning training set and special database 1 for their test set (LeCun 1998).

Handwritten digits found in special database 1 were gathered from high school students and it "contains 58,527 digit images written by 500 different writers" (LeCun 1998). The handwritten digits found in special database 3 were gathered from Census Bureau employees and this makes the data much cleaner and easier to recognize compared to special database 1 (LeCun 1998). NIST's special database "contain binary images (black and white pixels) of handwritten digits" (LeCun 1998).

MNIST dataset is a mix of NIST's special database 3 and special database 1. This has created a more independent and unbiased dataset. The MNIST training dataset has 60,000 records gathered from 250 writers and half of it is from NSIT special database 1 and the other half is from special database 3 (LeCun 1998). The MNIST test dataset has 10,000 records and 5,000 records are from special database 1 and the other half from special database 3 (LeCun 1998). MNIST training and test dataset uses different set of writers (LeCun 1998). We have used the entire MNIST dataset for this assignment.

As I mentioned earlier MNIST dataset is a subset of NIST database. Originally the black and white digit images from NIST "were size normalized to fit in a 20x20 pixel box and the digits aspect ratio were preserved" (LeCun 1998). "The resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm" (LeCun 1998).

For the MNIST dataset, the original images from the NIST database was computed into a 28x28 pixel image and the images were centered (LeCun 1998). This can be visualized or imagined as a large array of number, like the example shown in the image bellow (TensorFlow 2014). The NotMNIST dataset are also based 28x28 pixel image of letters.

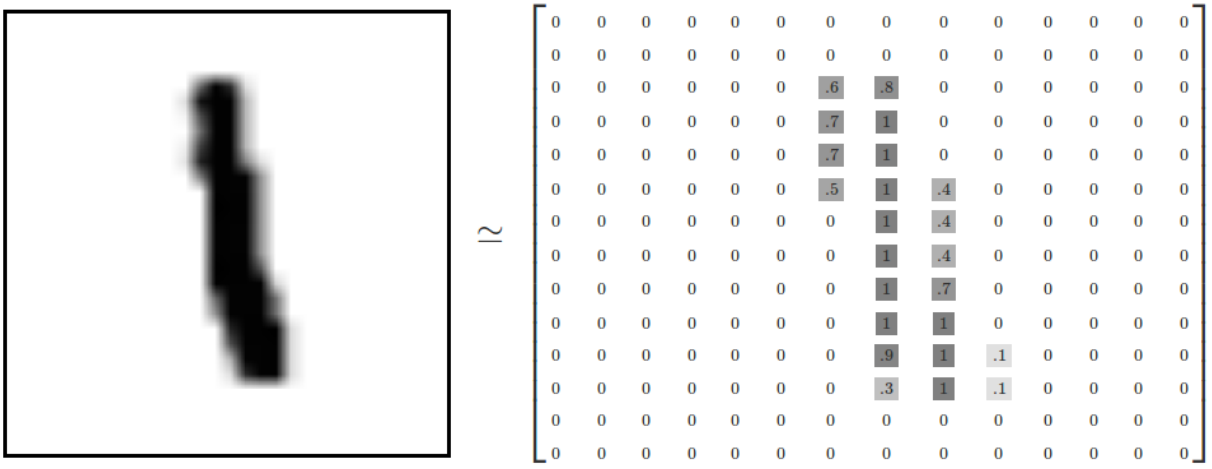


Figure above show a handwritten image, representing digit '1' and it can be visualized as a large array of numbers like shown on the right side. "The array of numbers can be thought

as describing how dark each pixel is" and it can be flatten into "vector of 28x28=784" (Olah 2014; TensorFlow 2014).

As you can see in the image above, the data points are between 0 – 1 to show the pixel intensity (TensorFlow 2014) but in the MNIST dataset that we are using for this assignment, the pixel intensity of data points range between 0 to 255. I'm only using the image above to understand the matter easily.

One of the pre-processing all the images in both MNIST and NotMNIST had to go through is converting grey scale images and converting into binary image using thresholding, undergone noise removal "using a median filter after thresholding process to remove the any clerical error" (Dalmia 2011; El Kessab et al. 2013; LeCun 1998). The MNIST and NotMNIST dataset also had to go through is size normalization and the digit in MNIST and the letters in NotMNIST had to be centered in a fixed size image (28pixel x 28 pixel). As you can see in the figure above, the background color for image '1' is white, so the associated value for the background is 0. This is the case for both MNIST and NotMNIST dataset. "Yann LeCun's MNIST dataset uses centering by center of mass within in a larger window" (LeCun 1998). This dataset used for this assignment had already undergone a lot of pre-processing before it become publically available on the Internet.

The MNIST and NotMNIST dataset used for the model is stored in CSV format contains "gray-scale images of hand-drawn digits, from zero through nine and letters between A to J" (Kaggle 2011). Each line on the CSV file represents one hand drawn image a digit for MNIST and a letter for NotMNIST. Each image is 28x28 pixel, so in total the image has 784 pixels and each pixel has a value associated with it. This associated value "indicate the lightness or darkness of the pixel" and the lower the number the lighter it is (*Digit Recognizer* 2011). Each pixel is represented by one 8-bit number, so the associated value is between 0 to 255.

The training dataset for MNIST and NotMNIST contains 785 columns/attributes and the test dataset contains 784 columns only. The first column in the NotMNIST training set contain the label for each image, telling the model which letter (between A to J) it is and the MNIST training set contains the label to tell which digit (between 0 to 9) it is. The other 784 columns represent the associated value for each pixel. The CSV file for MNIST training set looks like the figure bellow. NotMNIST CSV file has a similar format.

Col 1	Col 2	...		Col 108	...						
2	0	0	...	0	0	0	0	13	25	100	122	...
1	0	0	...	32	237	253	252	71	0	0	0	...
3	0	0	...	38	43	105	255	253	253	253	253	...
1	0	0	...	0	5	63	197	0	0	0	0	...
4	0	0	...	0	0	0	0	0	0	0	0	...
3	0	0	...	242	254	254	254	254	254	66	0	...
5	0	0	...	0	0	0	0	0	0	0	0	...
3	0	0	...	155	155	155	131	52	0	0	0	...
6	0	0	...	0	38	178	252	253	117	65	0	...
1	0	0	...	1	168	242	28	0	0	0	0	...
7	0	0	...	0	0	0	0	0	0	0	0	...
2	0	0	...	93	164	211	250	250	194	15	0	...

Figure shows a small portion of the CSV file for MNIST training dataset.

Yaroslav Bulatov created NotMNIST dataset and it's designed to be similar to MNIST dataset (Bui & Chang; Bulatov 2011). The idea behind this dataset is too create a similar dataset MNIST but to represent alphabets in different style. As I mentioned earlier, the NotMNIST dataset represents hieroglyphic fonts for letters between A to J and are not handwritten letters like the MNIST dataset (Flanagan 2015). What this means letters were generated using an assortment of fonts on a 28x28 grey-scale images (Tensorflow 2015) and this makes it much harder to recognizer a character or alphabet (Flanagan 2015).

NotMNIST dataset are also stored in a CSV file following the same format as MNIST dataset (as shown in the image above). So for NotMNIST dataset, there can only be 10 classes to classify the letters. The image bellow shows a very small of NotMNIST character images.

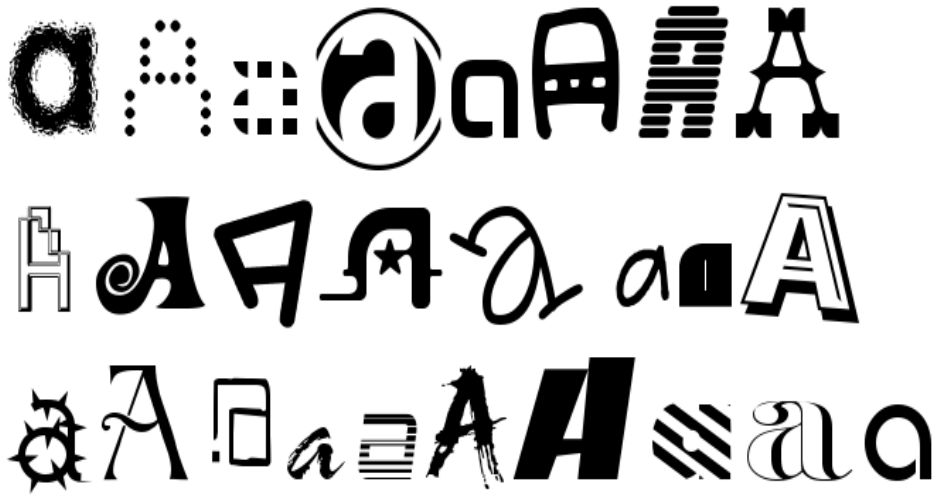


Figure above presents a small portion of images used in NotMNIST dataset for letter 'A' (Bulatov 2011).

As you can see, each image of 'A' is very different from the next one. This makes it much more challenge to classify NotMNIST dataset compared to MNIST and it's not as clean as MNIST dataset (Bui & Chang). "The NotMNIST dataset consists of small hand-cleaned part, about 19k instances, and large un-cleaned dataset, 500k instances" (Bulatov 2011).

A description of how you approached the problem, which algorithms you looked at and the parameter settings you used (10 pages).

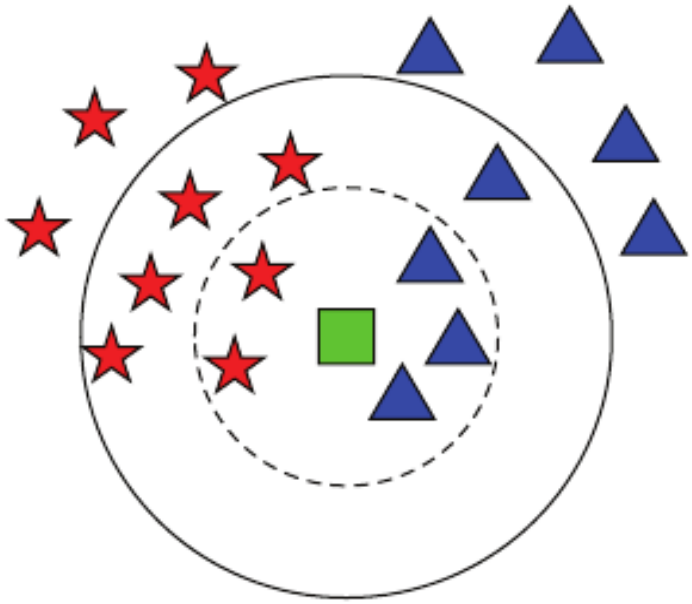
The problem here is to correctly to group handwritten images of digit between 0 to 9 in the MNIST dataset and to group the images of letters between A to J in NotMNIST dataset. NotMNIST dataset was created using hieroglyphic fonts.

This is a classification problem. The aim of classification is to recognize and understand the pattern that indicate the class to which each case belongs in the training dataset to predict the class for unlabeled or test dataset (Imandoust & Bolandraftar 2013). Therefore, a classification model will be built to analyze the labeled dataset to learn the predictive pattern (Imandoust & Bolandraftar 2013). This is also known as domain knowledge.

One of the learning algorithms we used to solve this problem is K-nearest neighbours. We choose K-nearest neighbours because of multiple reason. K-nearest neighbours is great effective for large datasets like MNIST and NotMNIST, "complex concepts can be learned by local approximation using simple procedures", it handles noisy data well and the cost of learning is zero (B´ejar 2013; people.revoledu.com 2015) . K-nearest neighbours is a simple classification technique that is easy to implement and debug (Cunningham & Delany 2007).

K-nearest neighbours is a non-parametric learning algorithm that can be used for both classification and regression. In a parametric model, "the hypothesis space is selected and a fixed set of parameters is adjusted to the training data. When we have a small amount of data it makes sense to have a small set of parameters and to constraint the complexity of the model (avoiding over-fitting)" (B´ejar 2013). However, over-fitting is not a problem when the dataset is large. A non-parametric model can't be categorized using a fixed of parameters, thus it falls under Instance Based Learning (B´ejar 2013). "Instance based learning is based on the memorization of data and the model is not associated with the learned concepts, thus the classification done by looking at the memorized example" (B´ejar 2013). In instance based learning, "the number of parameters is limitless and grows with the size of the data" (B´ejar 2013). This is also known as lazy learning.

K-nearest neighbours predict the class for the unlabeled data and the prediction is made based on majority votes of its neighbours and the unlabeled data are “assigned to the class most common among its k nearest neighbors (k is usually a small positive integer). If k = 1, then the data is simply assigned to the class of that single nearest neighbor” (Kaggle 2011). The next paragraph and diagram explains this is greater detail.



The figure above illustrates K-nearest neighbour’s classification (Wu et al. 2014).

As you can see in the figure above, the green square is test data that needs be classified either to the red stars class or into the blue triangle class. If k=5 (dashed lines circle), the green square test sample is classified into the blue triangle class because there are 3 blue triangles and only 2 red star inside the inner circle. Therefore it has 60% chances to be classified into blue triangle class compared to the red star class, which has a 40% probability (Wu et al. 2014). If k=10 (solid line circle) the green square test sample is classified as red star because there are 6 red stars and only 4 blue triangles in the outer circle. Therefore it has 60% chances to be classified into red star class compared to the blue triangle class, which has a 40% probability (Wu et al. 2014).

During the training stage, the MNIST and NotMNIST training dataset added to the K-nearest neighbours feature space as vectors and each vector have a class label. “Each feature in the dataset can be thought of as a dimension of the problem and each example is then a point in a n-dimensional feature space. K-nearest neighbours algorithm then

partitions the feature space when data sample is given and the boundaries also depend on the value of 'k', like shown in the figure above" (Settles 2003).

The 'k' value is a user-defined parameter, which can be decided during the classification stage. An unlabelled test sample is classified based on the most frequent k training samples that are nearest to the unlabelled test sample, like in the green square in the figure above. We used Euclidean distance metric in our model to calculate the distance between continuous variables, but in fact it was not really necessary for our model. This is because our model was made to classify images. Images have no physical distance between them, and our classifier only compared likeness between the two images, which means there is no actual distance to calculate. The larger the value of parameter 'k', the boundaries between the classes in the feature space become less distinct.

This is how K-nearest neighbours classification works. This principle is applied to our K-nearest neighbour's model built to classify the MNIST and NotMNIST dataset. Our model is built using python. Our python code is commented and it available under the appendix section of this report.

Artificial Neural Networks

After using k-nearest neighbors (KNN) algorithm we decided to run the data in the neural networks (NN) algorithm for number of reasons. The first reason is that it is not a true machine learning algorithm. What that means is that the algorithm will run through all the data every time choose the nearest neighbors then decide what this node should be based on them, but this algorithm will never be able to practice on what is the best number to choose for the voting five or seven for example, this algorithm will always take the given number of K compare to the nodes then decides based on the votes. In addition, the KNN algorithm if the pictures are not centered above each other the whole algorithm will be useless.

The Neural Network algorithm is a machine learning algorithm where this algorithm can learn and improve the output, and by manipulating the parameters in the algorithm we can enhance the results, a basic Neural Network algorithm has three components, one is activation function, and two is interconnections and finally learning rules.

There are couple of parameters in the NN algorithm, first we have Neurons, we can enhance the results by change the neurons either increasing them or decreasing them and observe the results then decide what is the best number of neurons to have its usually the more neurons the better, but there is a number if the neurons exceeded it the efficiency of the results will decrease too.

Second parameter are the layers, the number of layers can affect the results by adding more layers the results should improve. Third type of parameter is the learning rate which is the

parameter that controls the size of weight and bias changes in learning of the training algorithm.

The Fourth type of parameters is the momentum. Momentum what it does is adding a small fraction of the previous weight to enhance the current one. Momentum parameter is used to prevent the converging of the system to a local minimum or saddle point which causes the algorithm to get stuck in the wrong spot.

A high momentum parameter will also help to enhance the speed of the system convergence. However, if you set the momentum too high that can cause a risk of overshooting the minimum and that can cause the system to fail or to be useless. But if the coefficient of the momentum is too low then the momentum cannot reliably avoid local minima also that can lead to slow the system training.

Before going into explaining the whole algorithm it's important to explain the backpropagation algorithm. For our algorithm we decided to use the back propagation learning algorithm. A back propagation algorithm can be divided into three steps, forward propagation, back propagation, and weight update.

1. Forward propagation trains the input patterns through the Neural Network in the sake of generating the propagation output activation. Between every layer there are weights that are multiplied by the data and then run through an activation function.
2. Backward propagation output activations through the Neural Network using the target data in order to generate the deltas from multiplying the error and the derivatives from each of the activations.
3. From this change it's possible to update the weights to reflect how far off the weights are from predicting the final output

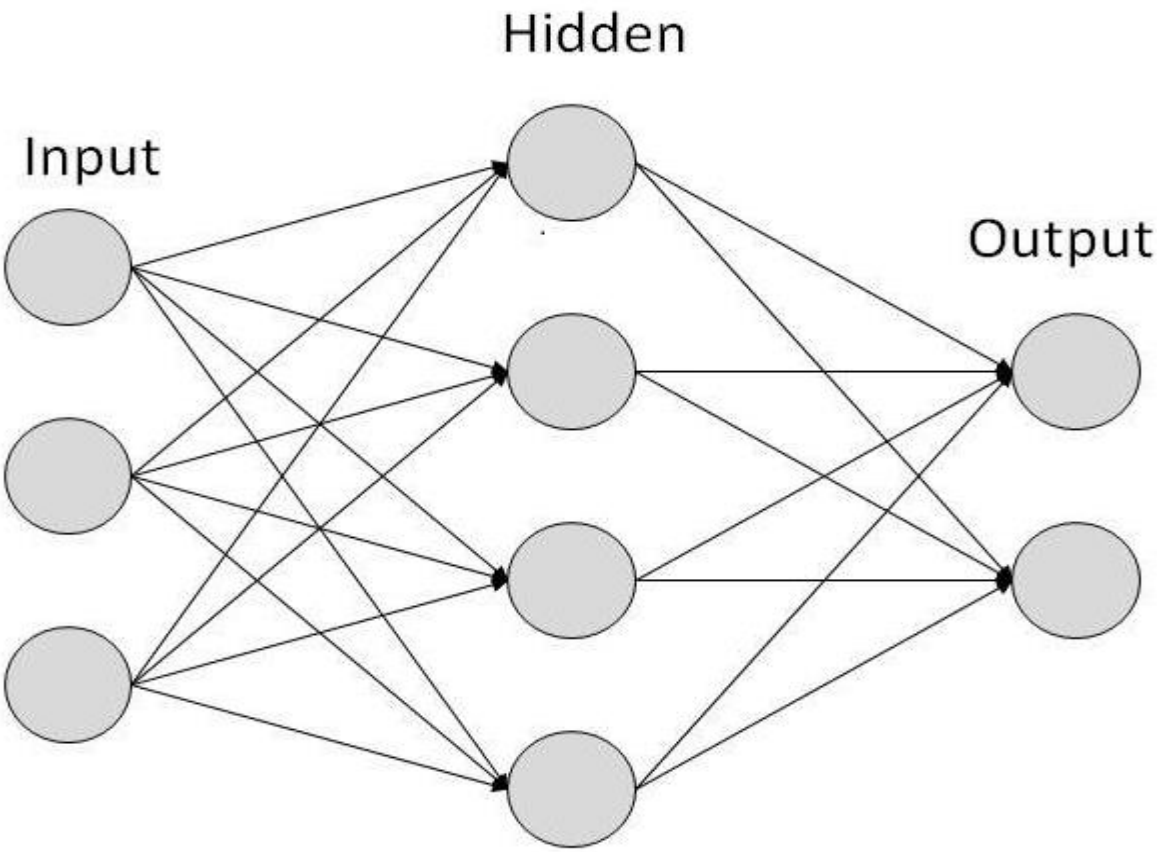
A learning rate is applied to the weight changes. The ratio the learning rate phase influences the quality and the speed of the learning process. When the ratio is large the neuron training will be faster, and when the ratio is low the training become more accurate when finished but will train slower. The slope here is important because the gradient of a weight illustrates if the error is increasing, this is why the weight must be updated in the opposite direction. The two steps above will be repeated until the network performance is satisfactory.

The back propagation should be explained before going into the whole Neural Network to ease the understanding of how the algorithm works, now I will explain the Neural Network step by step to the best of my ability.

Defining the model: Once the data is loaded the data can then be defined in our Neural Network model. Our model is ANN (artificial Neural Network). The mechanism of Artificial Neural Network was inspired from the way of how the human brain works.

Artificial Neural Network is composed of multiple nodes; these nodes imitate the neurons of human brain. The neurons interact with each other and they are connected by links. The nodes can take input data and perform some operations on the data. Then the result of the operations is moved to other neurons. The output at each node is called activation. In the ANN each link has a weight**.** ANN alters the weight values that what make it capable to learn. In the figure below I can show you a simple ANN.

Figure 1



After explaining the basics of an Artificial Neural Network it's possible to explain our code we used to train the MNIST dataset.

The following are some of the libraries we imported:

- Import math: This module provides access to the mathematical functions defined by the C standard.

- Import numpy as np: numpy is a powerful N-dimensional array object sophisticated functions tools. NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to integrate smoothly and quickly with a wide variety of databases.

- Import pandas as pd: **this** package providing fast, flexible, and expressive data structures designed to make working with relational data both smooth and intuitive. The aim of this package is to be a **real world** data analysis package.

First we insert the inputs as an array:

```
X = pd.read_csv ("mnist_train.csv", header=0)
X = X.as_matrix () #Converts to a numpy array
```

Then we insert the target data as y:

```
y = pd.read_csv("mnist_train_targets.csv", header=0)
```

But when we insert the dataset in pandas, we need to convert it into numpy so we insert the code below:

```
y = y.as_matrix ()
```

After convert from pandas to numpy, the flatten command it flat the lists, that means that it combine them into one flat list:

```
y = y.flatten ()
```

After that we need to convert the numbers from (1,2,3,4,5,….) into binary numbers zeros and ones in what is called “One Hot Encoding”:

```
y = pd.get_dummies(y).values
```

Then we create an instance of a Neural Network:

```
NN = NeuralNetwork (...)
```

Options in the instance can be passed to instance variables

- self.iterations = iterations (to repeat the mathematical procedure applied to the result of a previous application, typically after each iteration we are closer to the solution of a problem).

- self.learning_rate = learning_rate (in the training algorithm the learning rate controls the size of weight and bias changes).

- self.momentum = momentum (we use it to prevent the system from converging to a local minimum. A high momentum parameter can help to increase the system convergence speed. But if the momentum parameter is high that can cause the system to become unstable. And if the momentum coefficient is low that can slow down the training of the system).

- self.rate_decay = rate_decay (it makes the learning rate slow down to help the over fitting problem).

The weights can all be initialized as well

self.wIn = np.random.normal(loc=0, size=(self.input, self.hidden))
self.wIn = scale.fit_transform(self.wIn)

self.wOut = np.random.normal(loc=0, size=(self.hidden, self.output))
self.wOut = scale.fit_transform(self.wOut)

After the initialization we set up the arrays for activation, then create randomized weights, the next step is forward propagation. The feed forward algorithm loops over all the nodes in the hidden layer and adds together all the outputs from the input layer * their weights, the output of each node is the sigmoid function of the sum of all inputs, which is then passed on to the next layer.

After that comes the back propagation, for the output layer first it calculates the difference between output value and target value, second it get the derivative (slope) of the sigmoid function in order to determine how much the weights need to change, third it update the weights for every node based on the learning rate and sig derivative.
For the hidden layer first it calculate the sum of the strength of each output link multiplied by how much the target node has to change, second it get derivative to
Then we define test and run it just in feedforward, in here it just run the inputs to determine the weights.

After defining the forward propagation and the back propagation now they will be used in the training function to train our model. The second loop has I for input and t for target, in this loop we input an image every time and make adjustments each time.

The first loop is for the iterations, how many iterations we want to put or how much we want to train our data:

```
for i in range(self.iterations):
        iteration+=1
        error = 0.
        x=0
       For i, t in zip(inputs, targets):
          x+=1
          self.feedForward(i)
          error += self.backPropagate(t)
```

A similar function can be made for when after the training has been completed. The function can be used as a predict function, this function return list of predictions after training algorithm. It works the same as the one above but does not do the back propagation.

After that we have the sigmoid function

```
def dsigmoid(y):
   return y * (1.0 - y)
```

In this function we create a slope to have derivative, we used tanh over the sigmoid

```
def tanh(x):
   return math.tanh(x)
```

Here we find the derivative for tanh sigmoid

```
def dtanh(y):
   return 1 - y * y
```

That was a simple description of the Artificial Neural Network with the functions inside.

Convolutional Neural Networks

While Artificial Neural Networks are capable of functioning extremely well, they do have difficulties classifying images in datasets that are more difficult. For example, if the image is not centered or a different size, but also if the image looks quite different in shape. For example, the hand written digits below (Figure 2.1):

Figure 2.1

Images like in Figure 2.1, the notMNIST dataset, and things like Capchas, can be very difficult for Artificial Neural Networks to classify. To be able to classify such problems, a better technique needs to be employed to classify images that are more difficult in nature.

To be able to classify images such as those in Figure 2.1, a better solution such as Convolutional Neural Networks needs to be employed. Convolution Neural Networks were the ultimate goal of this assignment. We decided to build our Convolutional Neural Network on top of our Artificial Neural Network, after all there is no point in reinventing the wheel. In fact, the hidden layers of an Artificial Neural Network are still used in Convolutional Neural Networks. To create a Convolutional Neural Network, additional layers are created in front of the hidden layers.

There are three main tricks to getting a Convolutional Neural Network up and running. The first trick is of course convolution. The idea is to take what is called a receptive field and run it across an image many times, letting the receptive fields overlap. Do so would like something like below (Figure 2.2):

Figure 2.1

While this may seem strange at first, this is how a filter gets applied. A filter in a photo editing application works in the same way. It first divides the image up into receptive fields and then applies a matrix of numbers to every single receptive field. Specific types of matrices have different effects on images such as blurring and sharpening. Once the matrix is applied to all the receptive fields the pixels are averaged to make individual pixels. In Convolutional Neural Networks, this image is called a Feature Map. Convolutional Neural Networks use these filters but many of them. They create many filters starting with random weights. As the filter is applied the weight is run through the activation function during the feed forward phase. The back propagation can then be used to calculate the Delta and find the Loss to better adjust the weights of the filter. This creates a filter that can learn to manipulate the image until the algorithm can better predict what the end result is.

The next trick is in the activation function which is called ReLU which is short for Rectified Linear Units. This essentially works by simply turning all the negative numbers into 0s. The activation function in code looks like the following when using Python with NumPy during the forward propagation:

```
def relu(x):                #The matrix is passed as x
  return np.maximum(x, 0)   #The matrix is returned with 0s for negatives
```

The previous code works for the feed forward. For backpropagation, the derivative can be calculated like so:

```
def drelu(y):                       #The matrix is passed in as y
  return np.greater(y, 0).astype(float)  #The derivative is taken for values > 0
```

To finish calculating the Delta, the error can then be multiplied by the returned derivative:

```
convolution_derivative = drelu(convolution_activation)
convolution_delta = convolution_error * convolution_derivative
```

The last trick for Convolutional Neural Networks is something called Pooling. There are two different ways to do pooling, either by Average Pooling or Max Pooling. The most common is Max Pooling. Pooling works to shrink Feature Maps and the goal of this is to bring down the size but not lose the any discovered features in the Feature Maps. The idea is to select different windows inside the same image, but this time without overlapping. Either the average or the max value is taken and reconstructed to build a new smaller feature map than the one before. See Figure 2.3 below for an example of this:

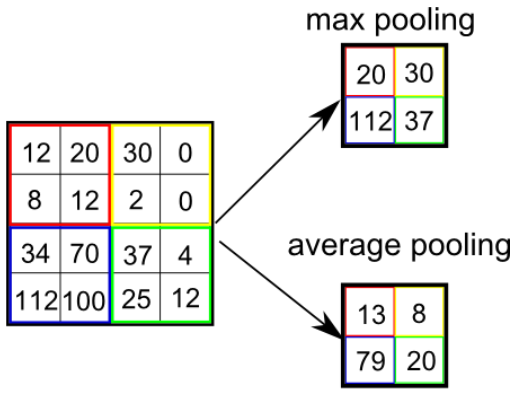


Figure 2.3

All of these layer can get stacked up in any order to try and achieve the best results. There are any number of ways to combine these layers and a lot of research goes into how to stack up these layers and set their parameters to achieve the best results. *See Figure 2.4.*

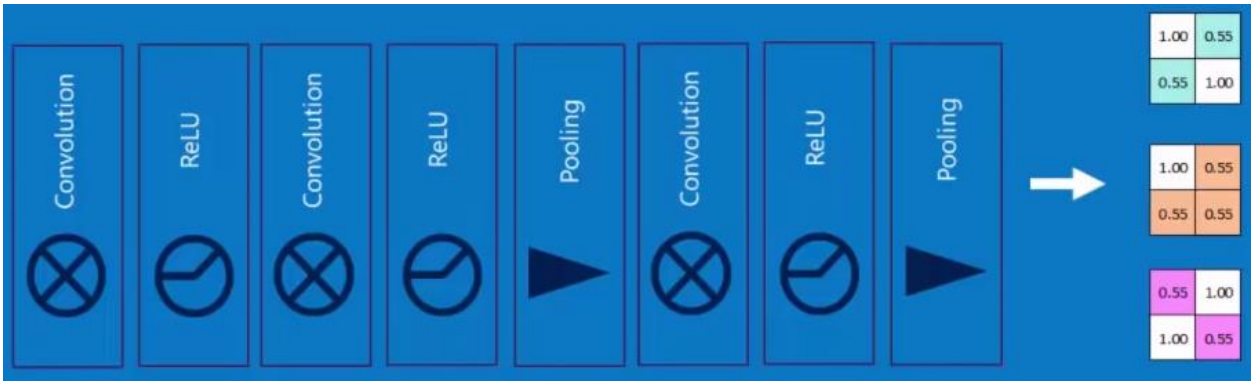
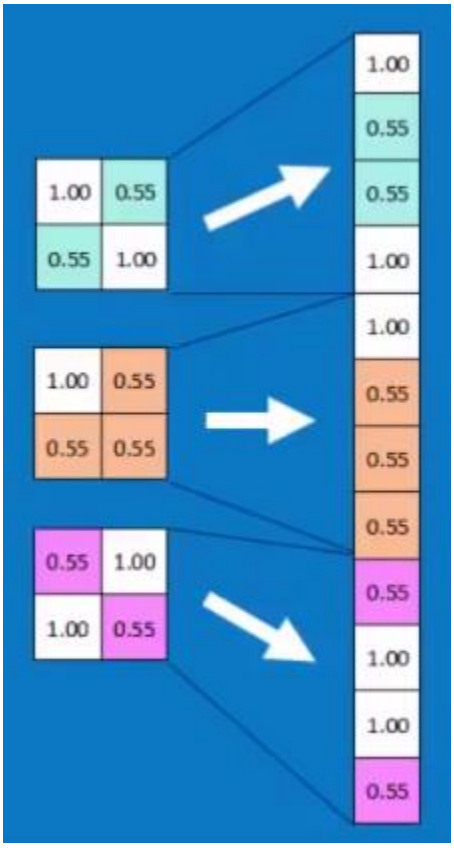


Figure 2.4



The final goal is to dilute feature maps down until they are finally small enough to extend out into what is called The Fully Connected Layer. This is what is called Feature Extraction. Once the Feature Extraction has been done far enough, it can be turned into one long list of features. This can be done by flattening a feature map that has been shrunken down to a feature map as small as 4 x 4 features. An example of this can be seen in Figure 2.5 on the left. The Fully Connected Layer is just one long list/array of features that was picked up while doing the convolutions up to this point.

The fully connected layer at the end further has more weights applied to it. In this way as all those features that got noticed before during the convolutional process, can be used to learn what points in the list apply to what outputs. The weights will be stronger at certain features for one particular number and be weaker at other points for others.

This doesn't have to be the end of the learning process. Any number of layers and neurons can be further added to improve the learning. The fully connected layer can be further connected to the next hidden layer, and so on. It is typical to have a convolutional neural network with additional hidden layers added along. In the example below (Figure 2.6), it's possible to see how

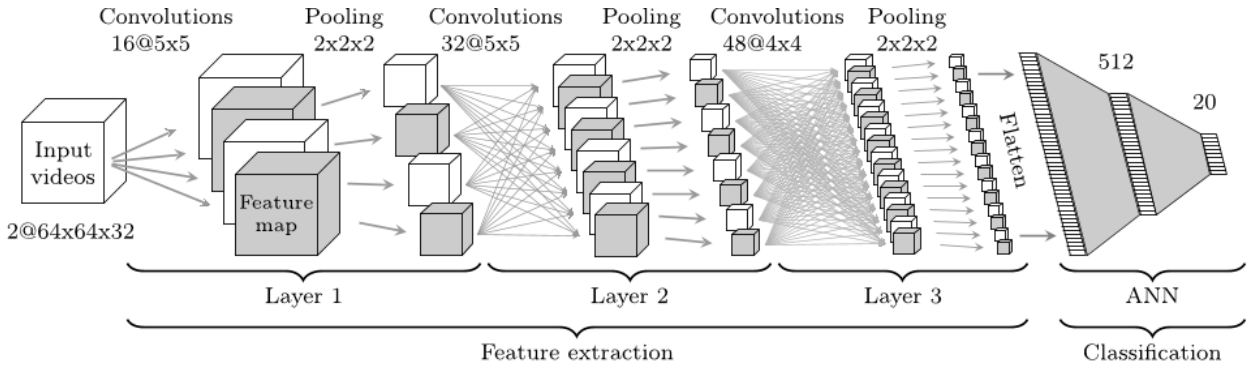


Figure 2.6

When we were working on our Neural Network we wanted to convert a regular Artificial Neural Network into a Convolutional Neural Network. Figure 2.6 shows how a Convolutional Neural Network is related to an Artificial Neural Network, and how it's possible to build one on top of the other. In this example, it has three Convolutional Layers which can be grouped into Feature Extraction. This is then passed into the actual

Classification part of the network which is essentially the traditional Artificial Neural Network. Sometimes Hidden Layers are called Fully Connected Layers in a Convolutional Neural Network. Like any other Artificial Neural Network, any number of layers and nodes can be used.

There are many parameters specific to Convolutional Neural Networks that can be adjusted. With regard to Feature Maps it's important to consider the size of the Receptive Field as well as what is called the Stride. Like was mentioned before, Receptive Fields overlap. Setting the Stride effects how much the Receptive Fields overlap. The Stride should remain less in sized as the Receptive Field to make sure that the overlap is maintained. Another important measure for the designer is to work out how many Feature Maps will be created. Then the designer can also decide how much to down sample using Pooling methods. When combining all these parameters, it has a direct effect on the number of layers that are made before reaching a Fully Connected layer, which is another important consideration. These can then be combined with the other parameters of the Artificial Neural Network such as Fully Connected Layers and number of nodes, and the learning. And like all Neural Networks, it's up to the designer to decide how many cycles/iterations/epochs the Neural Network to go through.

Recommended Classifier

Amongst the different classifiers we chose to examine for this problem, the classification method that would be recommend for solving digit recognition is the Convolutional Neural Network. In fact, Convolutional Neural Networks were specifically designed for image recognition, so recognizing hand written digits is very much what it was intended for (Flanagan 2015). In fact, in the world of Convolutional Neural Networks, this is a very easy classification problem for them.

In contrast, K-Nearest Neighbors was developed long before image recognition was a thing. Getting the algorithm to work with images is a little odd too. K-Nearest Neighbors usually classifies data points, but in this exercise we were not using data points but were using images. Euclidean distance could normally be used to see the nearest data points, but with images, it's completely unnecessary [2]. The loss function only needs to tell how similar the images are and pick k images that are most similar. Despite the simplicity of this algorithm it actually works extremely well. It does come with a few drawbacks.

It's not a true learning algorithm. K-Nearest Neighbors needs to run through the training set every single time it needs to make a classification. This is extremely wasteful in computing resources, and every classification would also need to keep the entire dataset with them. Furthermore, since the algorithm is only comparing how similar the images are, the still need human intervention to capture the digits. The digits need to be sized and centered in same sized images to be able to compare them. This makes it very problematic

for practical uses. Furthermore, the ability of the classifier to be improved is very limited. Our algorithm achieved around 96%. That is of course very good considering some of those digits are difficult for even humans to recognize, but it still means that perfecting the algorithm is just not possible.

The next algorithm we covered was the Artificial Neural Network. The biggest advantage of this algorithm of K-Nearest Neighbors is that it is a true learning algorithm. For one, this means that once the classifier is built, there is no more need to train the algorithm again, which saves on computing resources and also allows to classify data without needing the entire dataset every time it needs classification. Furthermore, Artificial Neural Networks allow for a multitude of techniques and parameters that can be manipulated to improve results. Indeed, Artificial Neural Networks using the Sigmoid function with two or three layers are capable of above 97%. Improvements can further be made when adding features such as rate decay, and using other activations functions such as Softmax. Using Artificial Neural Networks, the accuracy on MNIST data can be improved to the point where it can classify hand written digits as well as a human being can.

Convolutional Neural Networks improve on the shortcoming of Artificial Neural Networks. Convolutional Neural Networks allow the Neural Network to manipulate the images using a series of filters that help to further make sense of the images. The Convolutional Neural Network learns how to better filter an image to better interpret images that are harder to classify. This is a huge improvement over the classification of dataset such as the notMNIST dataset discussed in this paper. Furthermore, it is able to look through the image to find the feature it is attempting to classify, so images no longer need to be sized and centered by humans. This is why Convolutional Neural Networks are the basis for all state of the art classification techniques being used by data scientists right now.

Reference

B´ejar, J. 2013, *K-NN*, Universitat Politècnica de catalunya, viewed 1 October 2016, <http://www.cs.upc.edu/~bejar/apren/docum/trans/03d-algind-knn-eng.pdf>.

Bui, V. & Chang, L.-C., 'Deep Learning Architectures for Hard Character Classification'.

Bulatov, Y. 2011, *NotMNIST DATABASE* viewed 1 October 2016, <http://yaroslavvb.blogspot.com.au/2011/09/notmnist-dataset.html>.

Ciresan, D.C., Meier, U., Gambardella, L.M. & Schmidhuber, J. 2010, 'Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition', pp. 1-14.

Cunningham, P. & Delany, S.J. 2007, 'k-Nearest neighbour classifiers', *Multiple Classifier Systems*, pp. 1-17.

Dalmia, A. 2011, *K-NN Tutorial*, Indian Institute of Information Technology, viewed 1 October 2016, <https://researchweb.iiit.ac.in/~ayushi.dalmia/reports/Hand Digit Recognition.pdf>.

Digit Recognizer 2011, Kaggle, viewed 1 October 2016, <https://www.kaggle.com/c/digit-recognizer/data>.

El Kessab, B., Daoui, C., Bouikhalene, B., Fakir, M. & Moro, K. 2013, 'Extraction method of handwritten digit recognition tested on the mnist database', *International Journal of Advanced Science & Technology*, vol. 50.

Flanagan, D. 2015, *MNIST & NotMNIST Dataset Format*, GitHub, viewed 1 October 2016, <https://github.com/davidflanagan/notMNIST-to-MNIST>.

Imandoust, S.B. & Bolandraftar, M. 2013, 'Application of k-nearest neighbor (knn) approach for predicting economic events: Theoretical background', *International Journal of Engineering Research and Applications*, vol. 3, no. 5, pp. 605-10.

Kaggle 2011, *K-NN Tutorial*, Kaggle, viewed 1 October 2016, <https://www.kaggle.com/wiki/KNearestNeighbors>.

LeCun, Y. 1998, *MNIST DATABASE* Courant Institute, NYU, viewed 1 October 2016, <http://yann.lecun.com/exdb/mnist/>.
Olah, C. 2014, *Visualizing MNIST: An Exploration of Dimensionality Reduction*, GitHub.io, viewed 1 October 2016, <http://colah.github.io/posts/2014-10-Visualizing-MNIST/>.
people.revoledu.com 2015, *Strength and Weakness of KNN*, people.revoledu.com, viewed 1 October 2016, <http://people.revoledu.com/kardi/tutorial/KNN/Strength and Weakness.htm>.
Settles, B.H. 2003, *Feature Spaces*, university wisconsin-madison, viewed 1 October 2016, <http://pages.cs.wisc.edu/~bsettles/cs540/lectures/16_feature_spaces.pdf>.
TensorFlow 2014, *MNIST For ML Beginners*, TensorFlow, viewed 1 October 2016, <https://www.tensorflow.org/versions/r0.10/tutorials/mnist/beginners/index.html>.
Tensorflow 2015, *MNIST Deep Learning*, GitHub, viewed 1 October 2016, <https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/udacity/1_notmnist.ipynb>.
Winder, S.A.J. 2015, *Training neural nets on MNIST digits*, simonwinder.com, viewed 1 October 2016, <http://simonwinder.com/2015/07/training-neural-nets-on-mnist-digits/>.
Wu, J., Cui, Z., Sheng, V.S., Shi, Y. & Zhao, P. 2014, 'Mixed Pattern Matching-Based Traffic Abnormal Behavior Recognition', *The Scientific World Journal*, vol. 2014.

Appendix:

K-Nearest Neighbors:

```
import operator

import pandas as pd

import numpy as np



def knn_classify(image, trainingset, labels, k=5):

  N = trainingset.shape[0]

  baseImages = np.tile(image, (N,1))

  #print("The base shape is:")

  #show_image(baseImages[0])

  difference = baseImages - trainingset

  #print("The first trainingset images is:")

  #show_image(trainingset[0])
```

```
  #print("The difference is:")

  #show_image(difference[0])

  sqDifference = difference**2

  #print("The squared difference is:")

  #show_image(sqDifference[0])

  sqDistances = sqDifference.sum(axis=1)



  #print("Squared differences for the first number:", sqDistances[0])

  distances = sqDistances**0.5 #squareroot

  #print("The euclidean distance for the first one is:",distances[0])



  sorted_ind = distances.argsort()  #This sorts by index so that the order is preserved
for retreiving the labels

  #print("The lowest euclidean distance after sort is:", distances[sorted_ind[0]])

  #print("The max euclidean distance is:", distances[sorted_ind[N-1]])

  class_count={}

  for i in range(k):

      vote_i_label = labels[sorted_ind[i]]

      #print("The, ", i+1, " target number is: ", vote_i_label)

      class_count[vote_i_label] = class_count.get(vote_i_label,0) + 1 #Keeps track of
all of the votes in a dictionary

  #print("The counts of each vote: ")

  #print(class_count)

  sorted_class_count    =    sorted(class_count.items(),    key=operator.itemgetter(1),
reverse=True) #Sort by count

  return sorted_class_count[0][0] #Return the selection with the highest count



def             get_random_data(train_dir             =             "mnist_train.csv",
target_dir="mnist_train_targets.csv",n = 2001):

  dataset = pd.read_csv(train_dir, header=0)
```

```
    dataset = dataset.as_matrix()
    targets = pd.read_csv(target_dir, header=0)
    targets = targets.as_matrix()
    choices = np.random.choice(len(dataset), n)
    X = np.zeros(shape=(n,784))
    y = np.zeros(shape=(n))
    i = 0
    for choice in choices:
        X[i] = dataset[choice]
        y[i] = targets[choice]
        i += 1
    return X, y


def get_all_data(train_dir = "mnist_train.csv", target_dir="mnist_train_targets.csv"):
    dataset = pd.read_csv(train_dir, header=0)
    dataset = dataset.as_matrix()
    dataset = dataset.astype(np.float64)
    targets = pd.read_csv(target_dir, header=0)
    targets = targets.as_matrix()
    targets = np.reshape(targets, len(targets), 0)
    targets = targets.astype(np.float64)
    return(dataset, targets)


def get_test(test_dir = "mnist_test.csv", target_dir="mnist_test_targets.csv"):
    dataset = pd.read_csv(test_dir, header=0)
    dataset = dataset.as_matrix()
    dataset = dataset.astype(np.float64)
    targets = pd.read_csv(target_dir, header=0)
```

```
    targets = targets.as_matrix()
    targets = targets.astype(np.float64)
    return dataset, targets


def random_test(dataset, targets):
    choice = np.random.choice(len(dataset), 1)
    pic = dataset[choice]
    target = targets[choice]
    pic = pic[0]
    #print("The random choice was:", choice, "which is number,", target)
    return pic, target


def show_image(image):
  for j in range (1,28):
      row = []
      for i in range(1,28):
          if(image[(j-1)*28+i] >= 0 and image[(j-1)*28+i] < 75):
              row+=" "
          elif(image[(j-1)*28+i] >= 75 and image[(j-1)*28+i] < 150):
              row += "."
          elif (image[(j-1)*28+i] >= 150 and image[(j-1)*28+i] < 200):
              row += "*"
          else:
              row+="#"
      print (' '.join(map(str, row)))



n=1000 #How many tests to get total accuracy
```

```
#X, y = get_random_data(n=)      #Can change n, for number of variables
Xtrain, ytrain = get_all_data()  #Gets the whole dataset
Xtest, ytest = get_test()        #Gets a test image and target


right, wrong = 0, 0
for x in range(1, n):
   pic, target = random_test(Xtest, ytest)
   classifierResult = knn_classify(pic, Xtrain, ytrain , k=21)
   #print("The result from the classifier is:", classifierResult, )
   #print("The correct answer was: ", target)
   #show_image(pic)
   if (classifierResult == target):
       right += 1
   else:
       wrong+=1
print("THE FINAL ACCURACY IS:", 1 - (wrong/right))
```

Convolutional Neural Network:

CNN.py:

```
from math import sqrt
import numpy as np
import pandas as pd
import FowardProp as fp
import BackProp as bp
from sklearn.preprocessing import MinMaxScaler
from timeit import default_timer as timer
import time


class CNN(object):
   def __init__(self, iterations=3):
```

```
        self.iterations = iterations
        self.learning_rate = .01
        self.momentum = 0.5
        self.rate_decay = 0.0001

        #Initializes the inputs, hidden layers, and output
        self.input = 40 + 1               #i.e. How many pixels in an image.  1 is
added for for the bias
        self.hidden = 10                   #How many nodes in the hidden layer
        self.output = 10                   #How many possible outputs there are
        self.receptive_field = 81
        self.receptive_fields = 400
        self.filter = 10

        # set up array to store activation results
        self.aCon  = np.ones((self.receptive_fields, self.receptive_field))
        self.aIn    = np.ones(self.input)
        self.aHid    = np.ones(self.hidden)
        self.aOut    = np.ones(self.output)

        # create randomized weights
        # use scheme from 'efficient backprop to initialize weights
        scale = MinMaxScaler(feature_range=(-1, 1))

        self.filters = np.random.normal(loc=0, size=(self.filter,
self.receptive_field))
        self.filters = scale.fit_transform(self.filters)
        print ("FILTER WEIGHTS INITIALIZED:", self.filters.shape)

        self.wIn = np.random.normal(loc=0, size=(self.input, self.hidden))
        self.wIn = scale.fit_transform(self.wIn)
        print ("FULLY CONNECTED WEIGHTS INITIALIZED:", self.wi.shape)
```

```
        self.wOut = np.random.normal(loc=0, size=(self.hidden, self.output))
        self.wOut = scale.fit_transform(self.wOut)
        print ("OUTPUT WEIGHTS INITIALIZED:", self.wo.shape)


        # create arrays of 0 for changes
        # this is essentially an array of temporary values that gets updated at each
iteration
        # based on how much the weights need to change in the following iteration
        self.cCon  = np.zeros((400, 81))
        self.cIn = np.zeros((self.input, self.hidden))
        self.cOut = np.zeros((self.hidden, self.output))
        print("\nTHE NEURAL NETWORK HAS BEEN INITIALIZED...")


    def train(self, inputs, targets):
        print ("BEGIN THE TRAINING...")
        # N: learning rate
        for iteration in range(1,self.iterations):
            print("ITERATION:",iteration )
            start = timer()
            error = 0.
            x=0
            for i, t in zip(inputs, targets):
                x+=1
                fp.feedForward(self,i)
                error += bp.backPropagate(self,t)
            print("TOTAL ERROR FOR ITERATION,", iteration,", IS %-.5f" % error)
            # learning rate decay
            self.learning_rate = self.learning_rate * (
            self.learning_rate / (self.learning_rate + (self.learning_rate *
self.rate_decay)))


            vectoradd_time = timer() - start
            print("Iteration,", iteration, ",took ", vectoradd_time, "seconds")
        di.displayImage(self)
```

```
    def predict(self, X):
        prediction = fp.feedForward(self,X)
        return prediction


    def test(self, X, y):
        """
        Currently this will print out the targets next to the predictions.
        Not useful for actual ML, just for visual inspection.
        """
        for image, target in zip(X, y):
            print(target, '->', fp.feedForward(self,image))


def getTrainingData(train_dir = "mnist_train.csv",
target_dir="mnist_train_targets.csv"):
    dataset = pd.read_csv(train_dir, header=0)
    dataset = dataset.as_matrix()
    targets = pd.read_csv(target_dir, header=0)
    targets = targets.as_matrix()
    targets = targets.flatten()
    targets = pd.get_dummies(targets).values
    scale = MinMaxScaler(feature_range=(-1, 1))
    dataset = scale.fit_transform(dataset)
    print("THE TRAINING SET IS EXTRACTED...")
    return dataset, targets


def getTestData(train_dir = "mnist_test.csv", target_dir="mnist_test_targets.csv"):
    dataset = pd.read_csv(train_dir, header=0)
    dataset = dataset.as_matrix()
    targets = pd.read_csv(target_dir, header=0)
    targets = targets.as_matrix()
    targets = targets.flatten()
    targets = pd.get_dummies(targets).values
    scale = MinMaxScaler(feature_range=(-1, 1))
```

```
    dataset = scale.fit_transform(dataset)
    print("THE TEST SET IS EXTRACTED...")
    return dataset, targets


if __name__ == '__main__':
    Xtrain, ytrain = getTrainingData()
    Xtest, ytest = getTestData()


    CNN = CNN()


    CNN.train(Xtrain,ytrain)
    CNN.test(Xtest, ytest)
```

FowardProp.py

```
import Activations as a
import Convolutions as co
import numpy as np


def feedForward(self, inputs):


    #Convolution Layer One
    feature_maps  = co.convoluteOne(self, inputs)    #Gets the filtered receptive field
    max_pools = co.poolOne(feature_maps)             #Reduces the size of the images
using max pooling
    fully_connected = max_pools.reshape((40,))


    # preparing for the activation, gets all the inputs not including the bias
    for i in range(self.input - 1):     # -1 skips the bias
        self.aIn[i] = fully_connected[i] # the last element (bias) sits there unchanged
at the end in self.ai


    # hidden dot products and activations
    sum = np.dot(self.wIn.T, self.aIn) #Sum of all the weights to each layer
```

```
   self.ah = a.tanh(sum)            #Runs the layers through the activation


   # output dot products and activations
   sum = np.dot(self.wOut.T, self.aHid) #Sum of the layers to each ouput
   self.aOut = a.sigmoid(sum)         #Runs the ouput through the activation  #THINK
ABOUT ADDING OPTIONS TO CHANGE THIS
   return self.aOut                   #Returns the activated summed outputs
```

BackProp.py

```
import Activations as a
import numpy as np


def backPropagate(self, targets):


   # OUTPUT LAYER:
   output_error = -(targets - self.aOut)
   output_derivative = a.dsigmoid(self.aOut)
   output_deltas = output_error*output_derivative
   change = output_deltas * np.reshape(self.aHid, (self.aHid.shape[0], 1)) # update
the weights connecting hidden to output
   self.wOut -= self.learning_rate * change + self.cOut * self.momentum
   self.cOut = change


   # HIDDEN LAYER:
   hidden_error = np.dot(self.wOut, output_deltas)
   hidden_derivative = a.dtanh(self.aHid)
   hidden_deltas = hidden_error * hidden_derivative
   change = hidden_deltas * np.reshape(self.aIn, (self.aIn.shape[0], 1)) # update the
weights connecting input to hidden
   self.wIn -= self.learning_rate * change + self.cIn * self.momentum
   self.cIn = change


   #CONVOLUTIONAL LAYER:
   print "HIDDEN DELTA:", hidden_deltas.shape
```

```
    print "FILTER:", self.filters.shape
    con_error = np.dot(self.filters.T, hidden_deltas)
    con_derivative = a.drelu(self.aCon)
    con_deltas = con_error * con_derivative
    print con_deltas.shape, self.aCon.shape
    change = con_deltas * self.aCon
    #self.filters -= self.learning_rate * change + self.cCon * self.momentum
    self.cCon = change

    # calculate error
    error = sum(0.5 * (targets - self.aOut) ** 2)

    return error
```

Convolutions.py

```
import numpy as np
from sklearn.preprocessing import MinMaxScaler
import Activations as a
from PIL import Image

def convoluteOne(self, img):
    convolutions = np.zeros((400, 81))
    feature_map = np.zeros((400,))
    feature_maps = np.array(())
    scale = MinMaxScaler(feature_range=(0, 255))
    img = img.reshape((28, 28))
    img = scale.fit_transform(img)
    img = Image.fromarray(img)
    count = 0

    #Retrieve the receptive field from the training data
    for row in range (0, 20):
        for column in range (0, 20):
```

```
            temp = img.crop((column, row, column+9, row+9))
            temp = np.array(temp)
            temp = temp.reshape((81,))
            convolutions[count] = temp
            count+=1

    #The image needs to be back between 1 and -1 before applying the filter
    convolutions = np.array(convolutions)
    scale = MinMaxScaler(feature_range=(-1, 1))
    convolutions = scale.fit_transform(convolutions)
    #Apply the filter to the receiptive field
    for f in self.filters:
        for x in range(0, len(convolutions)):
            temp = convolutions[x].reshape((9,9))  #Cannot be single dimention for the
dot product
            tempf = f.reshape((9,9))               #Cannot be single dimention for the
dot product
            convolve = np.dot(temp, tempf)
            convolve = convolve.reshape((81,))
            self.aCon[x] = a.relu(convolve)        # Applies the relu activation
            feature_map[x] = np.sum(convolve)/81
        #img = feature_map.reshape((20,20))
        #img = Image.fromarray(img)
        #img.show()
        feature_maps = np.append(feature_maps, feature_map)
    feature_maps = feature_maps.reshape(10, 400)
    return feature_maps


def poolOne(feature_maps):
    pooled_featuremap = np.zeros((10,4))
    scale = MinMaxScaler(feature_range=(0, 255))
    count = 0
    for m in feature_maps:
        m = m.reshape((20, 20))
        m = scale.fit_transform(m)
```

```
        m = Image.fromarray(m)
        pixel = 0
        for row in range (0, 2):
            for column in range (0,2):
                temp = m.crop((column*10, row*10, (column*10)+10, (row*10)+10))
                temp = np.array(temp)
                temp = temp.reshape((100,))
                max = np.max(temp)
                pooled_featuremap[count][pixel] = max
                pixel+=1
        count += 1
    pooled_featuremap = np.array(pooled_featuremap)
    pooled_featuremap = pooled_featuremap.reshape((10, 4))
    scale = MinMaxScaler(feature_range=(-1, 1))
    pooled_featuremap = scale.fit_transform(pooled_featuremap)

    return pooled_featuremap
```

Activations.py

```
import numpy as np
import math

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def dsigmoid(y):
    return y * (1.0 - y)

def tanh(x):
    return np.tanh(x)

def dtanh(y):
    return 1 - y * y
```

```
def relu(x):
    return np.maximum(x, 0)


def drelu(y):
    return np.greater(y, 0).astype(float)
```