Chapter 15

Visualization

- Confusion Matrices
- PCA
- t-SNE

We frequently receive the following question about neural networks: "I've created a neural network, but when I train it, my error never goes to an acceptable level. What should I do?" The first step in this investigation is to determine if one of the following common errors has occurred.

- Correct number of input and output neurons
- Data set normalized correctly
- Some fatal design decision of the neural network

Obviously, you must have the correct number of input neurons to match how your data are normalized. Likewise, you should have a single-output neuron for regression problems or usually one output neuron per class for a classification problem. You should normalize input data to fit the activation function that you use. In a similar way, fatal mistakes, such as no hidden layer or a learning rate of 0, can create a bad situation.

However, once you eliminate all these errors, you must look to your data. For classification problems, your neural network may have difficulties differentiating between certain pairs of classes. To help you resolve this issue, some visualization algorithms exist that allow you to see the problems that your neural network might encounter. The two visualizations presented in this chapter will show the following issues with data:

- Classes that are easily confused for others
- Noisy data
- Dissimilarity between classes

We describe each issue in the subsequent sections and offer some potential solutions. We will present these potential solutions in the form of two algorithms of increasing complexity. Not only is the topic of visualization important for data analysis, it was also chosen as a topic by the readers of this book, which earned its initial funding through a Kickstarter campaign. The project's original 653 backers chose visualization from among several competing project topics. As a result, we will present two visualizations. Both examples will use the MNIST handwritten digits data set that we have examined in previous chapters of this book.

15.1 Confusion Matrix

A neural network trained for the MNIST data set should be able to take a handwritten digit and predict what digit was actually written. Some digits are more easily confused for others. Any classification neural network has the possibility of misclassifying data. A confusion matrix can measure these misclassifications.

15.1.1 Reading a Confusion Matrix

A confusion matrix is always presented as a square grid. The number of rows and columns will both be equal to the number of classes in your problem. For MNIST, this will be a 10x10 grid, as shown by Figure 15.1:

Figure 15.1: MNIST Confusion Matrix

```
> train.con
         predictions
targets    0    1    2    3    4    5    6    7    8    9
      0 1432    0    3    0    3    2    0    0    0    0
      1    3 1636    3    0    0    2    0    0    0    0
      2   50    1 1378    3   20    8    3    0    0    0
      3   22    1   49 1407    3   31    6    0    2    0
      4    2    0    5    5 1394   20    2    5    0    0
      5    2    0   14   27    5 1235   18    3   22    0
      6    0    0   23    0    7   41 1351    1    8    0
      7    0    0    0   13    6    3    0 1508    8   20
      8    0    0    4    2    2   21    8    2 1361   25
      9    0    0    0    0    0    2    8   26   21 1402
> |
```

A confusion matrix uses the columns to represent predictions. The rows represent what would have been a correct prediction. If you look at row 0 column 0, you will see the number 1,432. This result means that the neural network correctly predicted a "0" 1,432 times. If you look at row 3 column 2, you will see that a "2" was predicted 49 times when it should have been a "3." The problem occurred because it's easy to mistake a handwritten "3" for a "2," especially when a person with bad penmanship writes the numbers. The confusion matrix lets you see which digits are commonly mistaken for each other. Another important aspect of the confusion matrix is the diagonal from (0,0) to (9,9). If the program trains the neural network properly, the largest numbers should be in the diagonal. Thus, a perfectly trained neural network will only have numbers in the diagonal.

15.1.2 Generating a Confusion Matrix

You can create a confusion matrix with the following steps:

- Separate the data set into training and validation.
- Train a neural network on the training set.
- Set the confusion matrix to all zeros.
- Loop over every element in the validation set.

- For every element, increase the cell: row=expected, column=predicted.
- Report the confusion matrix.

Listing 15.1 shows this process in the following pseudocode:

Listing 15.1: Compute a Confusion Matrix

```
# x - contains dataset inputs
# y - contains dataset expected values (ordinals, not strings)
def confusion_matrix(x,y,network):
# Create square matrix equal to number of classifications
  confusion = matrix( network.num_classes, network.num_classes)
# Loop over every element
  for i from 0 to len(x):
    prediction = net.compute(x[i])
      target = y[i]
      confusion[target][prediction] = confusion[target][prediction
        ] + 1
# Return result
  return confusion
```

Confusion matrices are one of the classic visualizations for classification data problems. You can use them with any classification problem, not just neural networks.

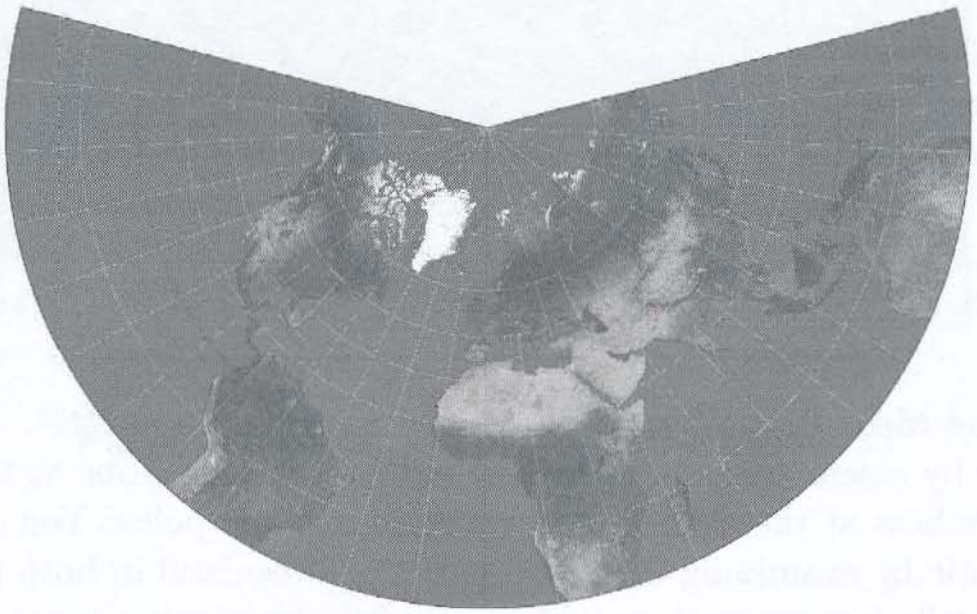
15.2 t-SNE Dimension Reduction

The t-Distributed Stochastic Neighbor Embedding (t-SNE) is a type of dimensionality reduction algorithm that programmers frequently use for visualization. We will first define dimension reduction and show its advantages for visualization and problem simplification.

The dimensions of a data set are the number of input (x) values that the program uses to make predictions. The classic iris data set has four dimensions because we measure the iris flowers in four dimensions. Chapter 4, "Feedforward Networks," has an explanation of the iris data set. The MNIST digits are images of 28x28 grayscale pixels, which result in a total of 784 input neurons (28 x 28). As a result, the MNIST data set has 784 dimensions.

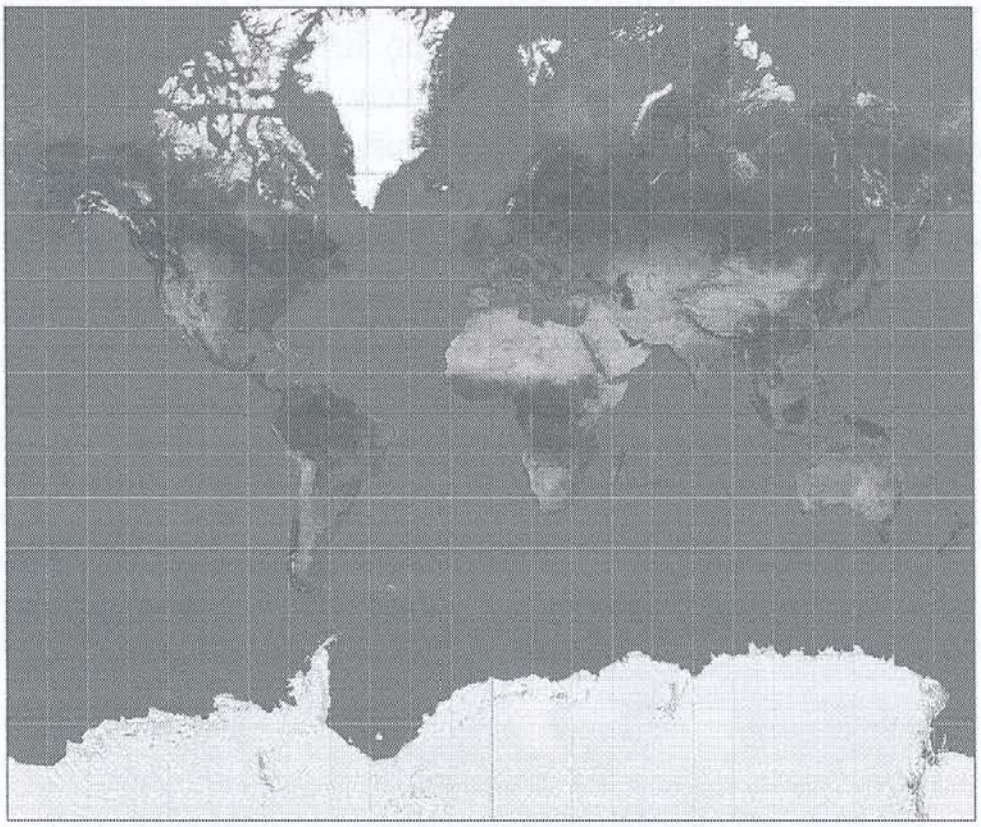
For dimensionality reduction, we need to ask the following question: "Do we really need 784 dimensions or could we project this data set into fewer dimensions?" Projections are very common in cartography. Earth exists in at least three dimensions that we can directly observe. The only true three-dimensional map of Earth is a globe. However, globes are inconvenient to store and transport. As long as it still contains the information that we require, a flat (2D) representation of Earth is useful for spaces where a globe will not fit. We can project the globe on a 2D surface in many ways. Figure 15.2 shows the Lambert projection (from Wikipedia) of Earth:

Figure 15.2: Lambert Projection (cone)



Johann Heinrich Lambert introduced the Lambert projection in 1772. Conceptually, this projection works by placing a cone over some region of the globe and projecting the image onto the globe. Once the cone is unrolled, you have a flat 2D map. Accuracy is better near the tip of the cone and worsens towards the base of the cone. The Lambert projection is not the only way to project the globe and produce a map, Figure 15.3 shows the popular Mercator projection:

Figure 15.3: Mercator Projection (cylinder)



Gerardus Mercator presented the Mercator projection in 1569. This projection works by essentially wrapping a cylinder about the globe at the equator. Accuracy is best at the equator and worsens near the poles. You can see this characteristic by examining the relative size of Greenland in both projections. Along with the two projections just mentioned, many other types exist. Each is designed to show Earth in ways that are useful for different applications.

The projections above are not strictly 2D because they create a type of third dimension with other aspects like color. The map projections can convey additional information such as altitude, ground cover, or even political divisions with color. Computer projections also utilize color, as we will discover in the next section.

15.2.1 t-SNE as a Visualization

If we can reduce the MNIST 764 dimensions down to two or three with a dimension reduction algorithm, then we can visualize the data set. Reducing

to two dimensions is popular because an article or a book can easily capture the visualization. It is important to remember that a 3D visualization is not actually 3D, as true 3D displays are extremely rare, as of the writing of this book. A 3D visualization will be rendered onto a 2D monitor. As a result, it is necessary to "fly" through the space and see how parts of the visualization really appear. This flight through space is very similar to a computer video game where you do not see all aspects of a scene until you fly completely around the object being viewed. Even in the real world, you cannot see both the front and back of an object you are holding–it is necessary to rotate the object with your hands to see all sides.

Karl Pearson in 1901 invented one of the most common dimensionality reduction algorithms. Principal component analysis (PCA) creates a number of principal components that match the number of dimensions to be reduced. For a 2D reduction, there would be two principal components. Conceptually, PCA is attempting to pack the higher-dimensional items into the principal components that maximize the amount of variability in the data. By ensuring that the distant values in high-dimensional space remain distant, PCA can complete its function. Figure 15.4 shows a PCA reduction of the MNIST digits to two dimensions:

Figure 15.4: 2D PCA Visualization of MNIST

The first principal component is the x-axis (left and right). As you can see, the matrix positions the blue dots (0's) at the far left, and the red dots

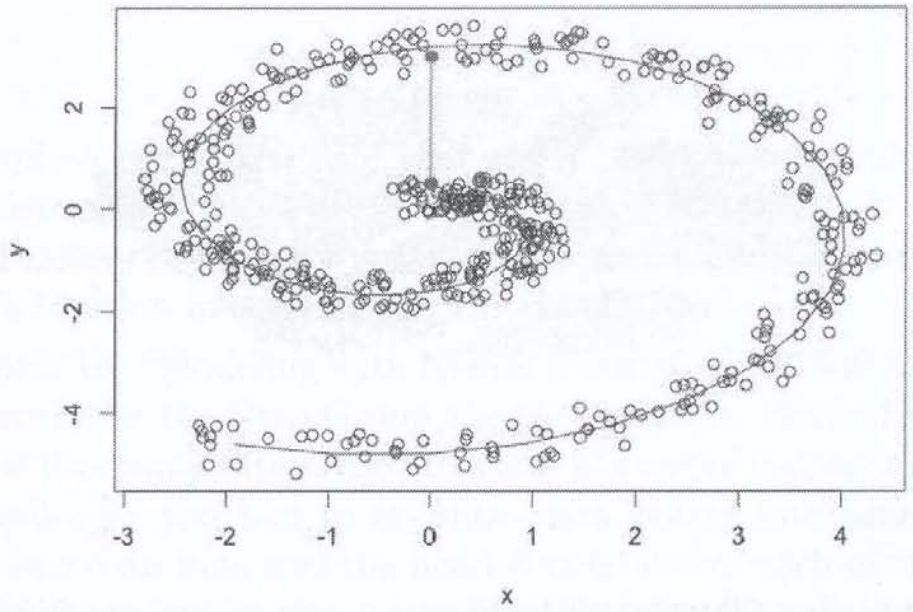
(1's) are placed towards the right. Handwritten 1's and 0's are the easiest to differentiate–they have the highest variability. The second principal component is the y-axis (up and down). On the top, you have green (2's) and brown (3's), which look somewhat similar. On the bottom are purple (4's), gray (9's) and black (7's), which also look similar. Yet the variability between these two groups is high–it is easier to tell 2's and 3's from 4's, 9's and 7's.

Color is very important to the above image. If you are reading this book in a black-and-white form, this image may not make as much sense. The color represents the digit that PCA classified. You must note that PCA and t-SNE are both unsupervised; therefore, they do not know the identities of the input vectors. In other words, they don't know which digit was selected. The program adds the colors so that we can see how well PCA classified the digits. If the above diagram is black and white in your version, you can see that the program did not place the digits into many distinct groups. We can therefore conclude that PCA does not work well as a clustering algorithm.

The above figure is also very noisy because the dots overlap in large regions. The most well-defined region is blue, where the "1" digits reside. You can also see that purple (4), black (7), and gray (9) are easy to confuse. Additionally, brown (3), green (2), and yellow (8) can be misleading.

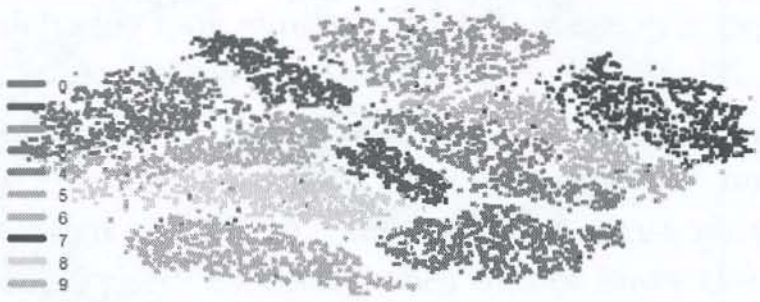
PCA analyzes the pair-wise distances of all data points and preserves large distances. As previously stated, if two points are distant in PCA, they will remain distant. However, we have to question the importance of distance. Consider Figure 15.5 that shows two points that appear to be somewhat close:

Figure 15.5: Apparent Closeness on a Spiral



The points in question are the two red, solid points that are connected by a line. The two points, when connected by a straight line, are somewhat close. However, if the program follows the pattern in the data, the points are actually far apart, as indicated by the solid spiral line that follows all of the points. PCA would attempt to keep these two points close as they appear in Figure 15.5. The t-SNE algorithm invented by van der Maaten & Hinton (2008), works somewhat differently. Figure 15.6 shows the t-SNE visualization for the same data set as featured for PCA:

Figure 15.6: 2D PCA Visualization of MNIST



The t-SNE for the MNIST digits shows a much clearer visual for the different digits. Again, the program adds color to indicate where the digits landed. However, even in black and white, you would see some divisions between clusters. Digits located nearer to each other share similarities. The amount of noise is reduced greatly, but you can still see some red dots (0's) sprinkled in the yellow cluster (8's) and cyan cluster (6's), as well as other clusters. You can produce a visualization for a Kaggle data set using the t-SNE algorithm. We will examine this process in Chapter 16, "Modeling with Neural Networks."

Implementations of t-SNE exist for most modern programming languages. Laurens van der Maaten's home page contains a list at the following URL:

http://lvdmaaten.github.io/tsne/

15.2.2 t-SNE Beyond Visualization

Although t-SNE is primarily an algorithm for reducing dimensions for visualization, feature engineering also utilizes it. The algorithm can even serve as a model component. Feature engineering occurs when you create additional input features. A very simple example of feature engineering is when you con-

sider health insurance applicants, and you create an additional feature called BMI, based on the features weight and height, as seen in equation 15.1:

$$BMI = \frac{\text{weight in kg}}{(\text{height in meters})^2} \tag{15.1}$$

BMI is simply a calculated field that allows humans to combine height and weight to determine how healthy someone is. Such features can sometimes help neural networks as well. You can build some additional features with a data point's location in either 2D or 3D space.

In Chapter 16, "Modeling with Neural Networks," we will discuss building neural networks for the Otto Group Kaggle challenge. Several Kaggle top-ten solutions for this competition used features that were engineered with t-SNE. For this challenge, you had to organize data points into nine classes. The distance between an item and the nearest neighbor of each of the nine classes on a 3D t-SNE projection was a beneficial feature. To calculate this feature, we simply map the entire training set into t-SNE space and obtain the 3D t-SNE coordinates for each feature. Then we generate nine features with the Euclidean distance between the current data point and its nearest neighbor of each of these nine classes. Finally, the program adds these nine fields to the 92 fields already being presented to the neural network.

As a visualization or as part of the input to another model, the t-SNE algorithm provides a great deal of information to the program. The programmer can use this information to see how the data are structured, and the model gains more details on the structure of the data. Most implementations of t-SNE also contain adaptions for large data sets or for very high dimensions. Before you construct a neural network to analyze data, you should consider the t-SNE visualization. After you train the neural network to analyze its results, you can use the confusion matrix.

15.3 Chapter Summary

Visualization is an important part of neural network programming. Each data set presents unique challenges to a machine learning algorithm or a neural network. Visualization can expose these challenges, allowing you to design

your approach to account for known issues in the data set. We demonstrated two visualization techniques in this chapter.

The confusion matrix is a very common visualization for machine learning classification. It is always a square matrix with rows and columns equal to the number of classes in the problem. The rows represent the expected values, and the columns represent the value that the neural network actually classified. The diagonal, where the row and column numbers are equal, represents the number of times the neural network correctly classified that particular class. A well-trained neural network will have the largest numbers along the diagonal. The other cells count the number of times a misclassification occurred between each expected class and actual value.

Although you usually run the confusion matrices after the program generates a neural network, you can run the dimension reduction visualizations beforehand to expose some challenges that might be present in your data set. You can reduce the dimensions of your data set to 2D or 3D with the t-SNE algorithm. However, it becomes less effective in dimensions higher than 3D. With the 2D dimension reduction, you can create informative scatter plots that will show the relationship between several classes.

In the next chapter, we will present a Kaggle challenge as a way to synthesize many of the topics previously discussed. We will use the t-SNE visualization as an initial. Additionally, we will decrease the neural network's tendency to overfit with the use of dropout layers.