Artificial Neural Networks:
their Training Process and Applications

Chaoyi Lou

Department of Mathematics
Whitman College

May 2019

Acknowledgement

I would like to express my deep gratitude to Professor Douglas Hundley, my project supervisor, for his patient guidance, enthusiastic encouragement, and useful suggestions of this project. I would also like to say thank you to Professor Pat Keef, for your advice and inspiration through the Senior Project class.

Abstract

Beginning in the 1800s, scientists intended to study the workings of the human brain. The concept of Artificial Intelligence developed over many years, and thanks to more advanced computers and spacious memories, now we can have 'Human-like' computer programs which can perform tasks for us. An Artificial Neural Network is one which learns from past data, and predicts for the future. Through this project, we not only gain a mathematical background of ANNs, but also touch base on those dealing with images, Convolutional Neural Networks.

Contents

List of Figures

Chapter 1

Introduction

Artificial Intelligence (AI) has been a new and popular topic in recent years. People try to design algorithms for computers so that they can accomplish tasks as close to real human beings. From Siri in Apple devices, Alexa built in smart homes, to self-driving cars in the near future, AI has become ubiquitous in our daily lives. In this project, we will take a close look at Neural Networks, which is a supervised algorithm belonging to a subfield of AI, called **Machine Learning**. Neural networks are used to learn from data, allowing them to do predictions on future, unseen data. Additionally, we will explore their functionality, their training process and advanced applications in several fields: health, finance, and visual recognition.

1.1 Natural vs. Artificial Neural Networks

An artificial neural network is a computer system modeled on the human brain and nervous system.

Figure 1.1 presents how a natural neural network works:

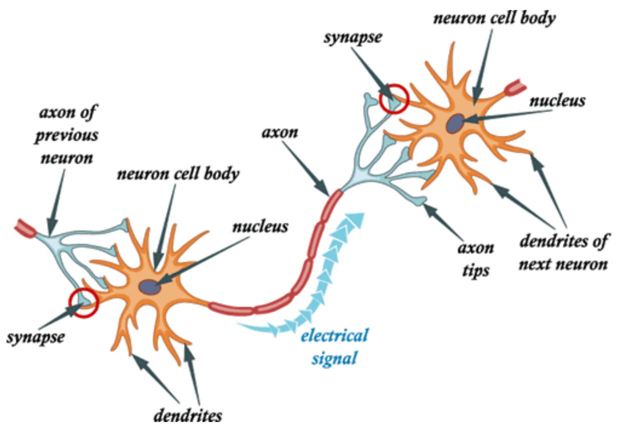


Figure 1.1: the Workflow of Natural Neural Networks

The impulse first travels through the **dendrites** of the neuron, which receive input and carry it towards a cell body. Then, the **cell body** interprets the input signal and transforms it into useful information. Finally, a single **axon** carries the output signal away.

However, the artificial neural networks (ANNs) cannot behave exactly the same as the natural ones, since what happens in the cell body is too complex to simulate. In essence, they have a similar structure to the natural ones. We use cold transistors and machines to build up the artificial system, and replace the inexplicable cell body by mathematical methods and computational algorithms which are responsible for analyzing and transforming the input.

By simulating the structure of natural ones, the artificial neural networks can recognize patterns and make decisions based on prior knowledge stored in the network. The main emphasis is how we construct the part in the artificial one which simulates the cell body in the natural one.

1.2 The Architecture of Artificial Neural Networks (ANNs)

A typical neural network has **neurons**, often called units or nodes. Their amount could be from a couple dozen to even millions and they are arranged in layers. All of the units can be classified into input units, hidden units, and output units, which connect the layers on either side.

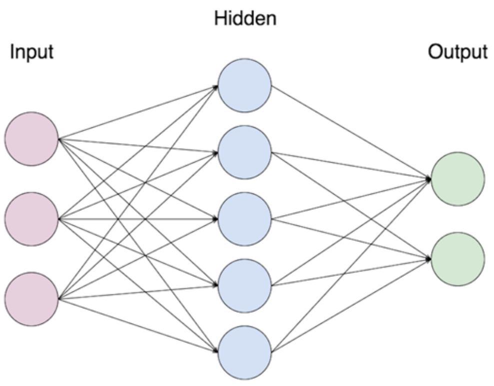


Figure 1.2: the Workflow of Artificial Neural Networks

In Figure 1.2, we can see that the input layer consists of input units, which are responsible for receiving numerical data from outside that the neural network attempts to learn about. The connection between one unit from input layer and one from hidden layer is represented by a number called a **weight**, which is denoted as W_i. The weight can be either positive or negative, which corresponds to the way actual brain cells excite or suppress others. If a unit has a higher corresponding weight, then it has more influence on the output. Initially, the weights are assigned at random and would be adjusted later through the training process.

There are two ways information can flow through a neural network. The **Feed-forward Neural Network** is the first and simplest type of ANNs. The data values go only in one direction, from the input units, to the hidden units, and finally to the output units. An example is shown below:

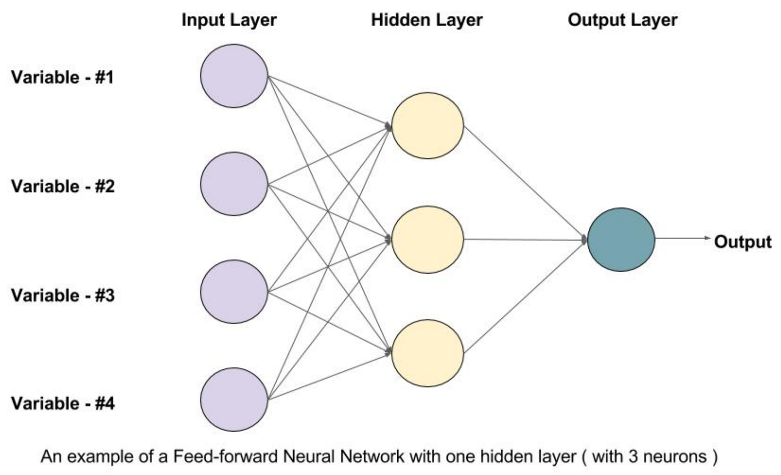


Figure 1.3: An example of a Feed-forward Neural Network

In Figure 1.3, the input layer collects the numerical variables from outside world and carries them to the next stage. No computation happens here.

Then, the data values arrive at the very first hidden layer, which has activation functions inside. The activation functions are responsible for performing computations and transferring the new values to the next hidden layer or output layer. While a feed-forward network will only have a single input layer and a single output layer, it can have zero or multiple hidden layers.

The output layer, which contains all output units, is responsible for carrying away the results computed from prior hidden layers.

The other type of neural network is the **Recurrent Neural Net-**

work. This kind of neural network has a loop or a cycle connecting all of the units inside of it. The recurrent attribute allows it to exhibit temporal dynamic behaviors.

Chapter 2

Mathematics Behind ANNs

We start from a single neuron to have a view from a mathematical perspective to better understand how ANNs are constructed and how they develop through the training process.

2.1 A Single Neuron

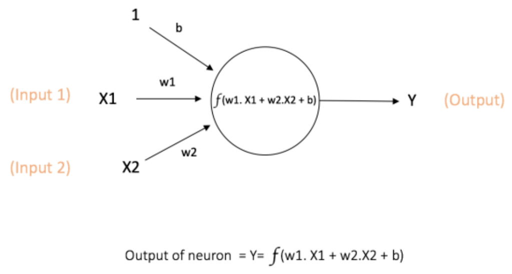


Figure 2.1: A single neuron of neural networks

Figure 2.1 shows a network consists of just one hidden layer containing one neuron. The single neuron receives input from the prior input layer, does computations, and sends the result away. We have two inputs here, x_1 and x_2 , with weights w_1 and w_2 respectively. The neuron applies a function f to the dot-product of these inputs, which is $w_1x_1 + w_2x_2 + b$. Besides the two numerical input values, there is one input value 1 with weight b, called the **Bias**. The main function of bias is to stand for unknown parameters or unforeseen factors.

The output Y is computed by taking the dot-product of all input values and their associated weights and putting it into the function f. This function is called the **Activation Function**.

We need activation functions because many problems take multiple influencing factors into account and yield classifications. For example, if we encounter a binary classification problem, the results would be either yes or no, so we need activation functions to map the results inside this range. If we encounter a problem involving probability, then we would wish to see our predictions from our neural network being in the range of $[0, 1]$. This is what activation functions can do for us.

There are two types of activation functions: linear activation functions and non-linear ones. The biggest limitation of linear ones is that they cannot learn complex function mappings because they are just polynomials of one degree. Therefore, we always need non-linear activation functions to produce results in desired ranges and to send them as inputs to the next layer. The following subsection will introduce several generally used non-linear activation functions.

2.1.1 Activation Functions

An activation function takes the dot-product mentioned before as a input and performs a certain computation on it. We put a certain activation function inside of neurons of hidden layers based on the range of the result we expect to see.

A notable property of activation functions is that they should be differentiable, because later we need this property to train the neural network using backpropagation optimization.

Here are some frequently used activation functions:

- **Sigmoid** or **Logistic**: takes a real-valued input and returns a output in the range [0,1]:

$$\delta(x) = \frac{1}{1+e^{-x}} \tag{2.1}$$

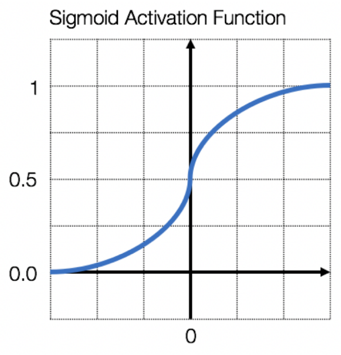


Figure 2.2: Sigmoid() Activation Function

In Figure 2.2, this is an S-shaped curve and the values going through the Sigmoid function will be squeezed in the range of $[0,1]$. Since the probability of anything exists only between the range of 0 and 1, Sigmoid is a compatible transfer function for probability.

Although the Sigmoid function is easy to understand and ready to use, we do not use it frequently because it has vanishing gradient problem. This problem is that, in some cases, the gradient gets so close to zero that it does not effectively apply change to the weight. In the worst case, this may completely stop the neural network from further training. Second, the output of this function is not zero-centered, which makes the gradient updates go far in different directions. Besides, the fact that output is in the narrow range $[0, 1]$ makes optimization harder. In order to compensate the shortcomings, $\tanh()$ is an alternative option because it is a stretched version of the Sigmoid function, in which its output is zero-centered.

- **tanh** or **hyperbolic tangent**: takes real-valued input and produces the results in the range [-1, 1]:

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.2}$$

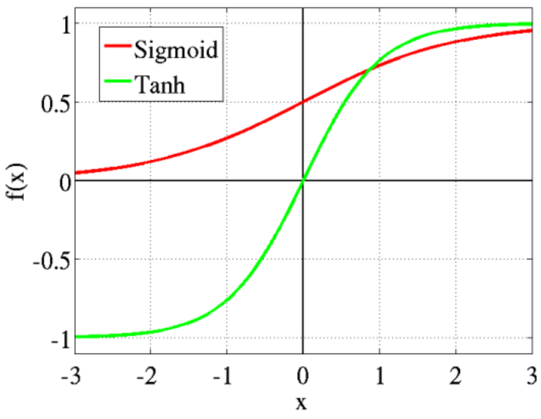


Figure 2.3: tanh() Activation Function

The advantage is that the negative input values will be mapped strongly negative and the zeros will be mapped near zero through this function. Therefore, this function is useful when we would like to perform a classification between two distinct classes. This function is preferred over the Sigmoid function in practice, but the gradient vanishing problem still exists. The following ReLU function rectifies this problem using a relatively simple formula.

- **ReLU** (Rectified Linear Unit): takes a real-valued input and replaces the negative values with zero:

$$R(x) = \max(0, x) \tag{2.3}$$

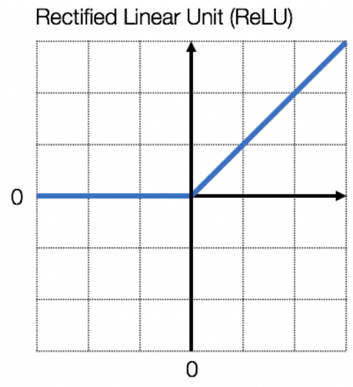


Figure 2.4: ReLU() Activation Function

The ReLU() activation function is trending now in the field of neural networks. It is used in almost all the convolutional neural networks or deep learning because it is a relatively simple and efficient function which avoids and rectifies the gradient vanishing problem.

The problem of this activation function is that all the negative values become zeros after this activation, which in turns affects the results by not taking negative values into account.

We use different activation functions when we know what characteristics of results we expect to see. Among these three activation functions, we usually start our training process using ReLU() because it works as a general approximator for most data sets.

2.1.2 Multi Layer Perceptron

There are two types of feed-forward neural networks:

- **Single Layer Perceptron**: the simplest feed-forward neural network with no hidden layers
- **Multi Layer Perceptron**: has one or more hidden layers which is useful for practical applications

We will focus on the multi-layer Perceptron because it can learn not only linear functions but also non-linear functions.

The feed-forward neural network in Figure 1.3 is an example of the multi-layer Perceptron. If we have a data set containing features and results, the multi-layer Perceptron will learn the relationship between features and results from the given data set, and predict the result for a new data point.

Generally in the input layer, we send n numerical inputs through n units:

$$\mathbf{x} = x_1, x_2, ..., x_n | \mathbf{x} \in \mathbf{R}^n$$

then we randomly assign weights for them at the first place:

$$\mathbf{w} = w_1, w_2, ..., w_n$$

The values for weights will be adjusted later in the training process for more accurate approximations.

In the hidden layer, we gather all the inputs by taking the dot product of **x** and **w**, and we call it the pre-state P:

$$P = w_1 \cdot x_1 + w_2 \cdot x_2 + ... + w_n \cdot x_n + b = \sum_{i=1}^{n} x_i w_i + b$$

We use vectors to represent them so that they can be viewed in matrix formatting. Figure 2.5 is an example of the calculation.

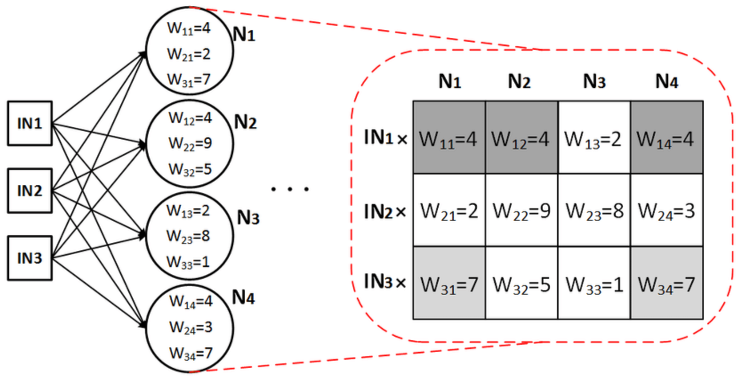


Figure 2.5: Getting Dot-products in Matrix Formatting

In Figure 2.5, the input layer has 3 units and the following hidden layer has 4 units. We can create a matrix of 3 rows and 4 columns and insert the values of each weight in the matrix as done above. This matrix would be called $\mathbf{W_1}$. We can do a matrix multiplication here and get a 1×4 pre-state matrix.

In the hidden layer of Figure 2.5, we have four pre-state N_i, each stores the dot-product of corresponding inputs and weights. These four pre-state values are ready to go through a certain activation function $\sigma()$. This is called the state S inside this hidden neuron:

$$S = \sigma(\mathbf{W}_i \mathbf{IN}_i + \mathbf{b}_i)$$

There could be more hidden layers following the first hidden layer, and the state(s) S, which store the transformed values, will be the input value(s) for the next layer. The matrix contains all the state values which will encounter the next weight matrix and produce new dot-products, and at that time, all the state values become the pre-state values of the next stage. The initial input values will go through every hidden layer in the neural net, repeat the same procedure mentioned above, and finally arrive at the output layer. The output values we receive are the ultimate state values in the very last hidden layer.

The above procedure explains how we set up our neural net. Now the question is, how to determine the weights? How to adjust the weights so that our neural net makes more accurate predictions? We need to know how to train our neural net based on the information we already have. The following subsection will introduce the error function in neural net, which is the key to determining the weights and biases.

2.2 General Model Building

A neural network learns from patterns of data and tries to make predictions as accurately as possible. Assume that we already have a set of p data pairs containing the variables and the results, $(\mathbf{x^{(1)}}, \mathbf{t^{(1)}})$, $(\mathbf{x^{(2)}}, \mathbf{t^{(2)}})$,...,$(\mathbf{x^{(p)}}, \mathbf{t^{(p)}})$ where $\mathbf{x^{(i)}}$ is input value and $\mathbf{t^{(i)}}$ is the target value for i=1, 2, 3, ..., p. We would like to build a neural net F so that ideally,

$$F(\mathbf{x^{(i)}}) = \mathbf{t^{(i)}}$$

However, typically we allow for error ϵ_i. Let $\mathbf{y^{(i)}}$ denote the output of the neural net so that

$$\mathbf{y^{(i)}} = F(\mathbf{x^{(i)}}) \text{ and } \mathbf{t^{(i)}} = \mathbf{y^{(i)}} + \vec{\epsilon_i}$$

We know that $\mathbf{y^{(i)}}$ depends on parameters, which are weights and biases, then it turns out as an optimization problem. We need to set up a neural net F that minimizes the error function, which is denoted below:

$$E = \frac{1}{N}\sum_{i=1}^{p} ||\mathbf{t^{(i)}} - \mathbf{y^{(i)}}||^2$$

where N is the number of training patterns. If it is a two-way classification problem, then $N = 2$. From this equation, we know that E is a function of the parameters in F, and we need to determine the values of weights that minimize the error by differentiating E.
If we focus on only one term of the sum, then

$$||\mathbf{t} - \mathbf{y}||^2 = (t_1 - y_1)^2 + (t_2 - y_2)^2 + ... + (t_p - y_p)^2$$

because we already know that the input and output values are fixed, and the only parameter here is the weight. We can differentiate both sides and get

$$\frac{\partial}{\partial W}(||\mathbf{t} - \mathbf{y}||^2) = -2(\mathbf{t} - \mathbf{y}) \cdot \frac{\partial \mathbf{y}}{\partial W}$$

Now we will be more specific and see how this fits in the neural net context. From a neural net, we have the output $\mathbf{y^{(i)}} = W_{ij}\mathbf{x^{(i)}} + \mathbf{b}$. We see that the output depends on the weight and if we differentiate both sides with respect to W_{ij} using chain rule, we get

$$\frac{\partial}{\partial W_{ij}}(||\mathbf{t} - \mathbf{y}||^2) = -2(t_i - y_i)x_j$$

where x_j is in the i^{th} coordinate position.
This derivative gives us the direction to the maximum, so in order to obtain the minimum point, we follow the opposite direction of this gradient. Additionally, we would like to see this derivative as close to 0 as possible in order to obtain the minimum of error. This algorithm follows the Widrow-Hoff Rule(see Appendix).

After figuring out which direction to go, we still need to know how far we go. We do not want it to move too slowly because we would like to finish this training part in an efficient manner. On the other hand, we do not want it to move a step too far; we may face the problem of not converging. Learning rate α is an important hyperparameter in gradient descent, because it determines how far each step should go. Unfortunately, we cannot analytically calculate a learning rate for a certain data set; we can know it only through trial and error. Typical values for a neural network with standardized inputs (or inputs mapped to the (0,1) interval) are less than 1 and greater than 10^{-6}.

Chapter 3

Training Session of ANNs

As previously stated, training a neural network means finding weights and biases that minimize the error function. There are several methods available to do this:

- Gradient Descent
- Newton's Method (an indirect approach, solving for where the derivative of the error is 0).
- Conjugate Gradient (Search along the eigenvectors of the Hessian of the error, the approach used in Matlab)

The Gradient Descent approach is mentioned in the general modeling section, and now we focus on how to apply this method to train the neural network to be more accurate. The process of updating the weights and reducing the error function is called the Backpropagation of Error.

3.1 Backpropagation of Error

We start from a simple 1-1-1 neural network, which contains an input x, two stages of weights, w_1, w_2, two stages of biases, b_1, b_2, an activation function $\sigma()$ and an output y.

$$x \to y = w_2\sigma(w_1 x + b_1) + b_2$$

Given a target t, the error of this neural net is

$$E(w_1, w_2, b_1, b_2) = \frac{1}{2}(t-y)^2 = \frac{1}{2}(t - (w_2\sigma(w_1 x + b_1) + b_2))^2$$

We want to minimize the error, so we move in the opposite direction of the gradient. Through the training process, we would like to update weights/biases in order to achieve a better error. Suppose we let u denote a generic parameter (either a weight or a bias). Using the gradient descent, u is updated by:

$$u_{\text{new}} = u_{\text{old}} - \alpha\frac{\partial E}{\partial u} = u_{\text{old}} + \alpha\Delta u$$

where α is called the learning rate, and the change in u is computed via the chain rule on the error. Notice that we incorporated the negative sign into Δu, because the derivative of the $(t-y)$ term will always be negative $t-y$. In particular,

$$\Delta u = -\frac{\partial E}{\partial y}\cdot\frac{\partial y}{\partial u} = -(t-y)\cdot -\frac{\partial y}{\partial u} = (t-y)\frac{\partial y}{\partial u}$$

Recall the pre-state P and state S mentioned in the prior section, we know that in this case $P = w_1 x + b_1$ and $S = \sigma(P)$. Now let us compute these partial derivatives for all the different parameters:

$$\begin{array}{ll} y = w_2 S + b_2 : & \\ \frac{\partial y}{\partial w_2} = S & \frac{\partial y}{\partial b_2} = 1 \\ \Delta w_2 = (t-y)S & \Delta b_2 = (t-y) \end{array}$$

And for the other parameters,

$$\begin{aligned} y = w_2\sigma(P) + b_2 : & \\ \frac{\partial y}{\partial w_1} = w_2\sigma'(P)\cdot x & \qquad \frac{\partial y}{\partial b_1} = w_2\sigma'(P) \\ \Delta w_1 = (t-y)w_2\sigma'(P)\cdot x & \qquad \Delta b_1 = (t-y)w_2\sigma'(P) \end{aligned}$$

From the example of a 1-1-1 neural net, we can generalize this to a three layer neural network in the form of $n-k-m$ with the activation function σ. Although we could define a different σ for every neuron, we typically use the same activation function for all the neurons in a single layer. Once that is done, we have to find matrices W_1, W_2 (and more, if we use more layers) and the bias vectors $\mathbf{b_1}, \mathbf{b_2}$.

Ideally, we would have much more data than that in order to get good estimates. In any case, the key idea is that once we have the derivative of the sum of squares error with respect to the weights, we can adjust the weights accordingly through the training process. In practice, the algorithm for reducing the error function is already stored in the neural network, so we only need to set up a neural network using a software (e.g. Matlab) and we can see the training process happens automatically.

Here is an example training session in Matlab:

```
P=-1:0.1:1;
T=sin(pi*P)+0.1*randn(size(P));
net=feedforwardnet(10);     %10 nodes in hidden layer
net=train(net, P, T)        %train the network
y=sim(net,P);               %get the output of the net
plot(P,T,P,y,'o')           %plot the data and the net output

tt=linspace(-1,1)           %new domain for the plot
yy=sim(net,tt);             %get the output from the net
```

```
plot(P,T,tt,yy,'k-')          %plot together...
```

This training session creates a 1x21 matrix P containing values in the range [0,1] and a 1x21 target matrix T to train the neural net. Using Matlab, we get the plot for the data and the net output:

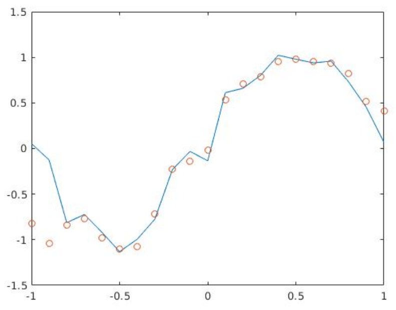


Figure 3.1: the Data and the Net Output Plot

To see it in a more clear way, we re-scale it and get:

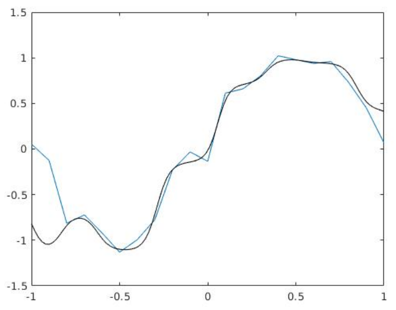


Figure 3.2: Re-scaled Plot for the training session

Figures 3.1 and 3.2 give visual representations of the actual output values(black line) and the target values(blue line). We conclude that the neural network does not create exactly the same results as the targets, but it acts as a general approximator.

We do not pursue a perfect matching in a neural net because we do not want to encounter an over-fitting problem, which allows the neural network to generalize better to unseen data. In order to get relatively accurate predictions, we need to consider separating the data set into several subsets used for different functionalities.

3.2 Data-sets Split

When training a neural network, we would split the data set we get into three subsets: the training data set, the validation data set, and the test data set in order to pursue a balanced point so that the neural net does not have an over-fitting problem and obtains a relatively high prediction rate.

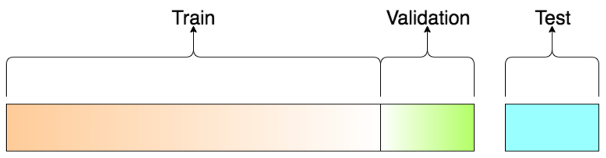


Figure 3.3: A visualization of the Split

- **Training Data Set**: The sample of data used to train the neural net.
- **Validation Data Set**: The sample of data used to provide an unbiased evaluation of a neural net fit on the training data-set while tuning the hyper-parameters[1].
- **Test Data Set**: The sample of data used to provide an unbiased evaluation of a final version of neural net fit on the training data-set.

We use the training data set to train the weights and biases in our neural net. The neural net learns from this data set.

The validation set is used to evaluate a neural net, but this is for the frequent evaluation. We use the validation set results and update higher level hyper-parameters. Therefore, the validation set in a way affects a model, but indirectly.

The test data set is used when the neural net is completely trained. It contains carefully sampled data that spans the various classes which the model would face when used in the real world.

By utilizing these three data sets, we know how to avoid the over-fitting problem because there would be a point before that in which the performance of the training goes down while those of the validation and test sets go up.

1. the variables which determine the network structure(E.g. Number of Hidden Units) and the variables which determine how the network is trained(E.g. Learning Rate). They are set before training.

Chapter 4

Applications of ANNs

Artificial Neural Networks are utilized as prediction tools in many fields. At this point, we will go through two examples, one in Health and another one in Finance. ANNs are really powerful tools to do predictions because they repeatedly train themselves to reach a proper prediction point, unlike those traditional statistical tools which are only applied once and no adjustment happens during the modeling process.

4.1 An Application in Health using Matlab

We use the built-in pattern recognition application in Matlab to train a neural network for a breast cancer data set from the University of California Irvine(UCI). This neural network classifies cancers as either benign or malignant depending on the characteristics of sample biopsies. The input is a 9×699 matrix defining nine attributes of 699 biopsies, and the target data set is a 2×699 matrix where each column indicates a correct category with one in either benign or malignant. The nine attributes are Clump thickness, Uniformity of cell size, Uniformity of cell shape, Marginal Adhesion, Single epithelial cell size, Bare nucleoli, Bland chomatin, Normal nucleoli, and Mitoses.

4.1.1 The Implications behind the Performance

The following figure shows the best validation performance point:

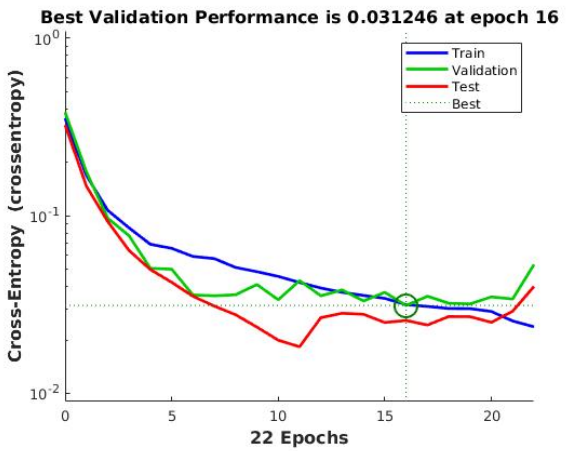


Figure 4.1: Performance of the neural network for the breast cancer dataset

We do a performance check to avoid the over-fitting problem and we can see in Figure 4.1 that the performance of the training set goes down, which is the main reason we do not want to continue our training.

4.1.2 Visualization of the results: Confusion Matrix

A confusion matrix is a very useful visualization tool built in Matlab because it is a figure that is used to describe the performance of a classification model (or "classifier") on a set of test data for which the true values are known. From confusion matrices, we can have a clear view of how many false-positive and true-negative results exist in the neural network.

The following figure shows four confusion matrices:

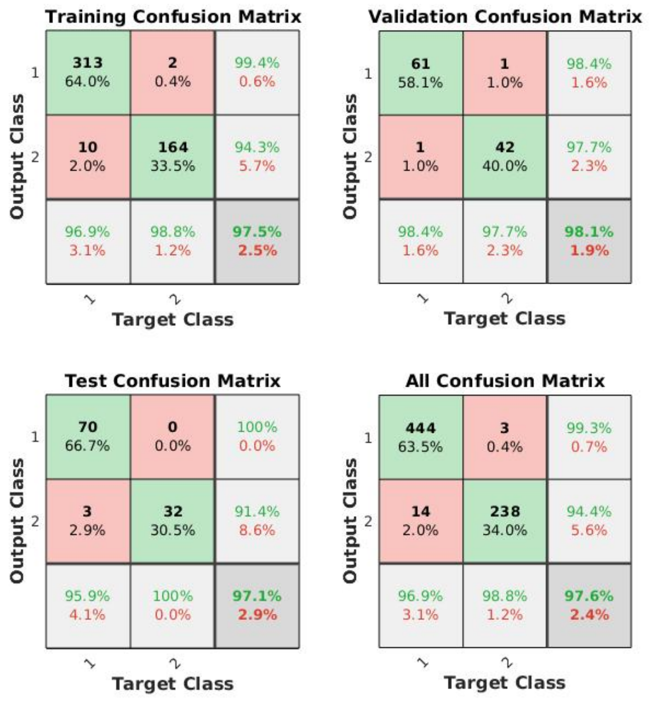


Figure 4.2: Confusion matrices produced by neural network pattern recognition

In the training confusion matrix, we know that the neural net puts 313 people in the right position - benign group, and puts 2 people in the wrong group when they should be in the benign group. The correct rate is 99.4%. It also makes right predictions for 164 people in the malignant group and wrong predictions for 10 people in the benign group, while in fact they should be in the malignant group. The correct rate is 94.3%. We interpret the confusion matrices for the validation set and test set, and we can conclude that this neural network experiences relatively accurate training and has a good performance with an overall correct rate of 97.6%.

4.2 Using ANNs to Manage Credit Risk

A credit risk is the risk of default on a debt that may arise from a borrower failing to make required payments. Analysts conduct an assessment called **Credit Risk Analysis** before deciding whether to sign a financial contract with people, so they can identify the obligors and quantify the amount to repay their borrowing well in advance. Companies need to make prediction about whether a person would pay his/her bill on time by examining different factors in order to plan things ahead.

4.2.1 Methods used in Credit Risk Analysis

There are two ways for credit risk modeling:

1. Data Mining or Statistical Learning Approach
2. Natural Computing and Mathematical Modeling

The key metrics in credit risk modeling are credit rating (probability of default), exposure at default, and loss given default.

When analysts calculate the risk ratings, which are classification and regression tree problems that either identify a customer as "risky" or "non-risky", they tend to use classical statistical modeling to predict the classes based on past data. However, nowadays artificial neural networks become an option to deal with the classification problem, because they are more flexible and capable of modeling complex non-linear problems.

Analysts find that most of the applicants are non-defaulting in credit assessment, and only a small number of them are defaulters. The relatively small sample size of bad credit people in the application process indicates there would be more people in the good category than in the bad category, which is extremely imbalanced. This imbalance results in performance degradation, which makes the predictive modeling more challenging. However, an artificial neural network which leverages clustering and merging can achieve balanced data, so that it can better decide whether an applicant should be granted a loan or not. Several methods can be used to balance the input data, and a frequently used one, **k-means**, is included in the Appendix.

Although Neural Networks are more powerful than traditional models, we may encounter problems if we choose a wrong data set. If some unrelated variables are involved, the performance of neural networks would show no improvement compared to traditional tools. In the next section we will go through an example demonstrating how important data pre-processing and selection are, and what would be the performance of neural networks if we get irrelevant variables involved in the neural net.

4.2.2 Should this Loan be Approved or Denied?

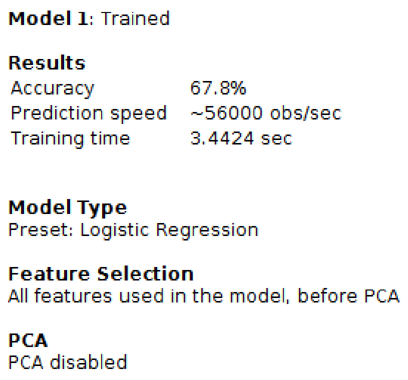
The example we would like to examine is “Should This Loan be Approved or Denied?”: A Large Data set with Class Assignment Guidelines(Li et al.,2018). In this article, the authors intend to "assume the role of loan officer at a bank and are asked to approve or deny a loan by assessing its risk of default using logistic regression"(Li et al.,2018).

Logistic regression is a statistical method to do predictive analysis, which is equivalent to the neural network with no hidden node(Zhang et al. 1999). Based on the data set provided, we will set up a neural network to see its performance and compare it to the traditional logistic approach.

The study gives us a holistic view of how to do predictive analysis, but several methods used in the study need further considerations. First, the study uses a large and rich data set which contains 2102 observations from the U.S. Small Business Administration(SBA). There are 27 variables from SBA, and it chooses five predictors:

1. **New**(=1 if NewExist=2 (New Business), =0 if NewExist=1 (Existing Business))
2. **RealEstate**(=1 if loan is backed by real estate, =0 otherwise)
3. **DisbursementGross**(in Dollars)
4. **Portion**(Proportion of gross amount guaranteed by SBA)
5. **Recession**(=1 if loan is active during Great Recession, =0 otherwise)

The problem is that this study chooses predictors through a discussion instead of a serious analysis based on formal methods: "..., we provide the students with the “National SBA” dataset, a background of the SBA, and the assignment with its learning objectives. Since economic models should be based on sound economic theory, we engage students in a discussion which requires them to identify which explanatory variables they think would be good indicators or predictors of the potential risk of a loan: likelihood of default (higher risk) versus paid in full (lower risk)"(Section 4.1, Li et al.,2018). This is a biased approach, because it directly eliminates the chance to include some actually important explanatory variables, and the selection at this point influences later analysis. As stated in this study, the predictive analysis involving logistic regression finally results in 67.8% accuracy, which is shown in Figure 4.3.



Model 1: Trained

Results

Accuracy	67.8%
Prediction speed	~56000 obs/sec
Training time	3.4424 sec

Model Type
Preset: Logistic Regression

Feature Selection
All features used in the model, before PCA

PCA
PCA disabled

Figure 4.3: Logistic Model Information

If we take a close look at the chosen predictors, we will see that three out of five variables are categorical, which means that they are either 0 or 1. Because the selection of predictors is biased, and the relationship between them does not go through examination, it is reasonable to doubt that the model would do better if choosing better variables. However, at this moment, we would like to see whether neural networks would do better while keeping the same input data set.

Now, we can set up a neural network for this problem involving the same five variables and see the performance using Matlab. Input is a 2102×5 matrix, representing static data: 2102 samples of 5 elements(predictors), and the target is a 2102×1 matrix, representing static data: 2102 samples of 1 element(decisions:proved or not). The confusion matrices for this data set is presented below:

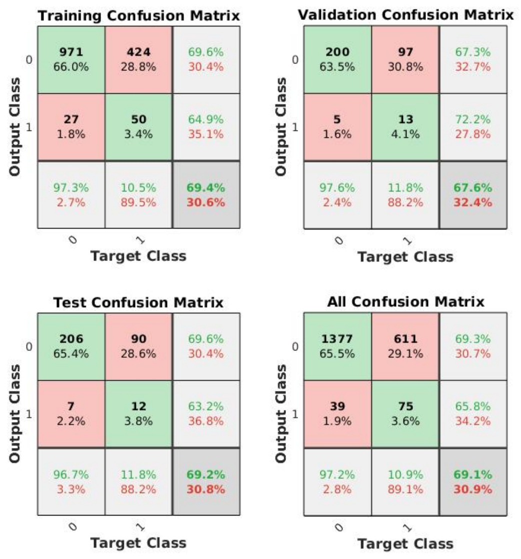


Figure 4.4: Confusion Matrices for Credit Analysis

The overall accuracy does not differ much from that of the logistic regression prediction.

This example gives us some insights about the core determinant of the accuracy of a neural network. Data selection is the most difficult and controversial part prior to the model. Before passing the data into the neural network, we need to limit a data set size, thus accelerating the training time. For example, typically in data selection we first obtain little improvement of the accuracy if we remove some data, but the accuracy will drop if we continue to remove more. There are many methods available for us to do the data selection, and if we would like to set up our own neural network in the future, the data selection is the crucial part we need to put much more emphasis on.

Chapter 5

Convolutional Neural Network

In deep learning, a convolutional neural network is a class of deep neural networks, most commonly applied to analyzing visual imagery. Recall that in previous sections, the artificial neural networks accept numerical inputs. Now the convolutional ones can process images, because all digital images are represented by pixels, which are numerical in nature.

If we are dealing with a black and white image, we can see that certain pixels are totally white, while some are gray and black. White pixels are represented with the value 0 and black ones are represented with a value of 255. For color images, the pixels are evaluated in three channels: red, green and blue(commonly known as RGB).

We can evaluate every color image in these three color densities and here is a visualization of a parrot picture:

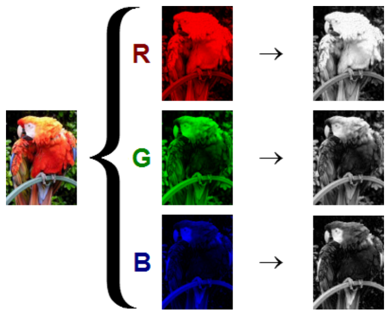


Figure 5.1: A RGB Channels Separation

In contrast, a gray image has only two dimensions, which is a little bit easier to handle.

Now the question is, what we can do using CNNs? The major functionality is **object recognition**. There are three types of them: Image Classification, Object Detection, and Instance Segmentation.

In Image Classification, the input to the problem is an image and the required output is simply a prediction of the class that the image belongs to.

Object Detection and Instance Segmentation are more advanced tasks. The input is still the same, but the output is more specific. In Object Detection, the required output consists of bounding boxes surrounding the detected objects; In Instance Segmentation, the output is a pixel grouping that corresponds to each class.

We will focus on the image classification to see how CNNs work.

5.1 Building Blocks of CNNs

In Image Classification, we will face a challenge that the appearance of objects is dynamic. There are infinite number of ways objects can appear in an image. This makes it difficult for traditional image classification techniques because storing an infinite number of pictures in a computer is still difficult to achieve.

However, we do not need to prepare images with different characteristics at all. Humans differentiate objects into different categories based on certain features. A toddler can easily differentiate cats and dogs once she has seen just few of them. Instinctively, people look for features; and yes, we can also teach a computer to look for those certain features within the entire image. The key lies in **convolution**.

There are two concepts we need to understand, filtering and convolution. We will explain them through an example.

5.1.1 Filtering and Convolution

Suppose we have a 9×9 image as our input, and we need to identify the image as an X or an O. There are numerous ways to write them, but they have totally different characteristics. We know that Os tend to have flat horizontal edges, while Xs tend to have diagonal lines.

A common shared characteristic of an object is called a **filter**, which is responsible for identifying the object.

As in Figure 5.2, the filter is of size 3×3, and the presence of this characteristic gives us a big hint on the class of the image. If an image contains a horizontal edge, then the image is probably an O.

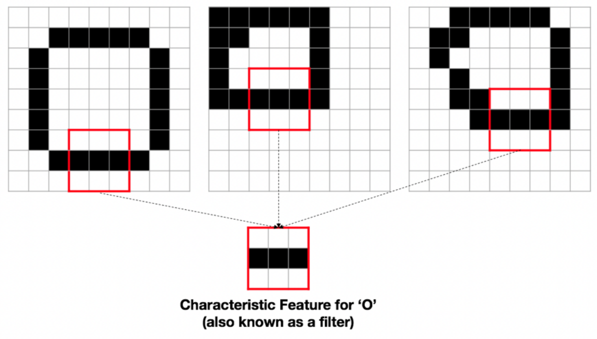


Figure 5.2: A filter of O

For Xs, the filter would be a 3×3 figure containing diagonal lines:

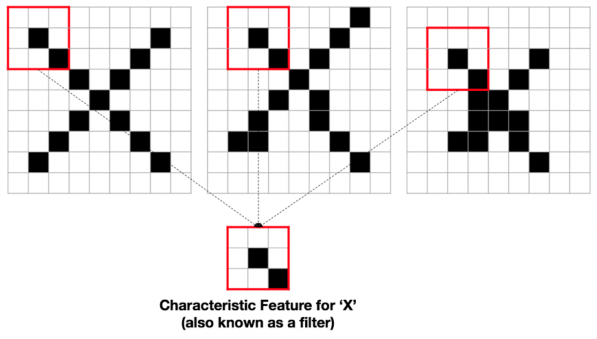


Figure 5.3: A filter of X

After defining the filters, we need to search for them in the input images. We simply perform a search taking the 3×3 filter and sliding it through every single pixel in the image to find a match. The mathematical function performed by the filter is the element-wise multiplication of the sliding window with the filter.

For simplicity, we assume pixel values are 0 or 1, instead of the real intensity values in the range of [0,255]. From Figure 5.4, we can see that the filtered value is 2 for this window. Then we slide this window

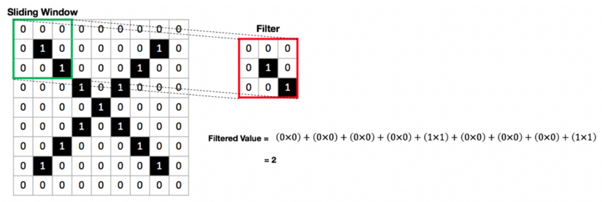

Sliding Window
Filter
Filtered Value = (0×0) + (0×0) + (0×0) + (0×0) + (1×1) + (0×0) + (0×0) + (0×0) + (1×1)
= 2

Figure 5.4: Filtering Operation on the top left-hand corner

toward the right to check the next 3×3 section in the image. We can see the filtered value for next section is 0.

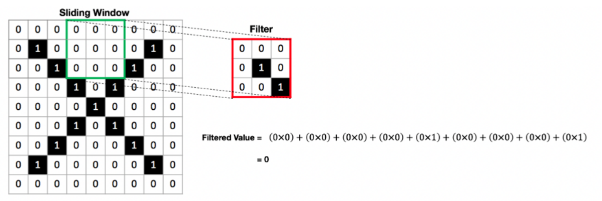

Sliding Window
Filter
Filtered Value = (0×0) + (0×0) + (0×0) + (0×0) + (0×1) + (0×0) + (0×0) + (0×0) + (0×1)
= 0

Figure 5.5: Filtering Operation on the next section

The process of sliding the window through the whole image and calculating the filtered value is known as **convolution**. Generally, the neural net performs convolution layer before all the layers in the artificial net. Filtering and convolution provide us with information of areas which contain the characteristics features. This equips our neural net to be able to do dynamic object recognition just like a human being.

Note that there are two main hyperparameters in a convolutional layer: the filter size and the number of filters. In the example we just used one filter, but we can increase the number of filters to find more characteristics. The next layer just after the convolutional layer is max pooling layer.

5.1.2 Max Pooling

This layer is responsible for reducing model complexity and avoiding over-fitting by reducing the number of weights after each convolutional layer.

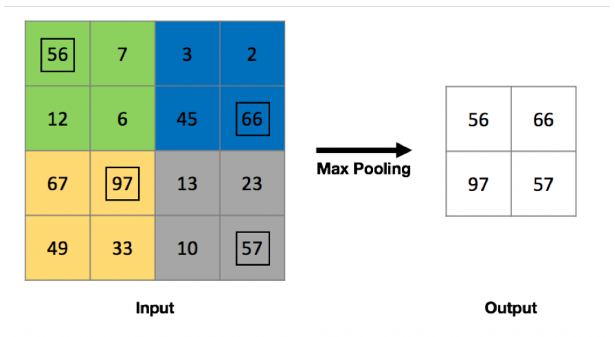


Figure 5.6: An Example of Max pooling

In Figure 4.6, we have an 4×4 input matrix and a 2×2 max pooling layer. Max pooling simply looks at each 2×2 region of the input, and retains the maximum number of that region. Therefore, the dimensionality is reduced. This is a very effective approach especially when we encounter a huge amount of input data.

5.1.3 CNN Model Building

Now, we need to put all things together and construct a complete Convolutional Neural Network(CNN).

When a CNN receives input images, it will first target the characteristics and reduce the complexity, which would be done by the convolutional layer and max pooling layer. This pair of layers would be repeated twice, and their main purpose is to identify characteristic features. The next two fully connected layers are responsible for learning how to make predictions, just like multi layer Perceptrons mentioned in prior sections.

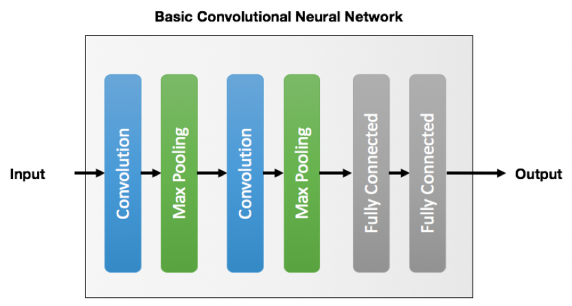


Figure 5.7: Basic Structure of CNNs

Essentially, the whole process is automatic, because the early layers learn and identify features on their own, and the later layers are self-trained to do predictions. The implication is significant. Instead of handcrafting features for the machine learning algorithms, as we did in two earlier ANN examples, we are simply providing all the data to the CNN as it is. There is no data selection needed. Everything happens automatically and we get the results in different classes.

5.2 Image Classification Using CNNs

Now that we understand the theory behind CNNs, we can set up a convolutional network using Python to classify cats and dogs. This image dataset is provided by Microsoft, which contains 12499 images in each class. Due to the massive amount of images, it may take a long time to classify by hand. However, through a CNN, the images would be put into two classes in just 10 minutes. There is one important package named **Keras** in Python. We will use several built-in functions in Keras to set up our CNN. The functions are listed in the following code:

```
from keras.models import Sequential
from keras.layers import Conv2D,MaxPooling2D
from keras.layers import Dropout,Flatten,Dense
from keras.preprocessing.image import ImageDataGenerator
model = Sequential()
```

5.2.1 Setting up Hyper-parameters

Before adding any convolutional layers, we need to set some hyper-parameters:

1. **Convolutional layer filter size**: Most modern CNNs use a filter size of 3×3.
2. **Number of filters**: it is a number we can customize, but more filters would take more time to process, while a low number of filters would result in low accuracy.
3. **Max pooling size**: a common max pooling size is 2×2.
4. **Batch size**: the number of training samples to use in each mini batch during gradient descent. A large batch size results in more accurate training, but longer training time and memory usage.
5. **Epochs**: how many times the training data goes through the model.

We have the following hyper-parameters set up for the dogs and cats data set:

```
FILTER_SIZE=3      %a general filter size
NUM_FILTER=32      %set as 32, a good balance between
                     speed and performance
```

```
INPUT_SIZE=32       %set as 32, good for speeding up the
                      training
MAXPOOL_SIZE=2      %usually 2X2
BATCH_SIZE=16       %set as 16
EPOCHS=10           %each training set will be passed to
                      the model 10 times
```

5.2.2 Setting up Layers

Now we can add our first convolutional layer and max pooling layer, with 32 filters, each of size 3×3:

```
model.add(Conv2D(NUM_FILTERS, (FILTER_SIZE, FILTER_SIZE),
                 input_shape = (INPUT_SIZE, INPUT_SIZE, 3),
                 activation = 'relu'))

model.add(MaxPooling2D(pool_size=(MAXPOOL_SIZE, MAXPOOL_SIZE)))
```

The Conv2D function sets up the convolutional layer for us, and the activation function used is the most commonly used one, ReLU(). The MaxPooling2D function is responsible for reducing the complexity. We will repeat to add these two layers again after them.

Before we move to the fully connected layers, we need to flatten its input, so that the multidimensional input values could be transferred into single dimensional values. We achieve this by running the following code:

```
model.add(Flatten())
```

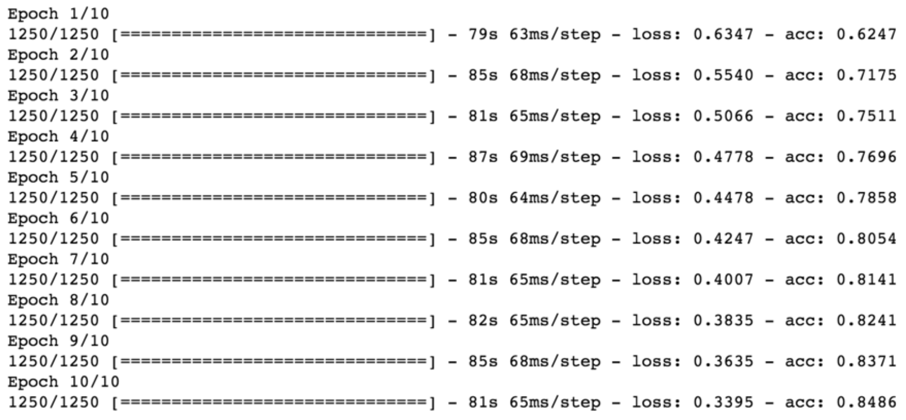
We can now add two fully connected layers and run the CNN:

```
model.add(Dense(units = 128, activation = 'relu'))
model.add(Dropout(0.5))    %reduce overfitting by randomly
                              setting 50% of the input to 0
model.add(Dense(units = 1, activation = 'sigmoid'))
model.compile(optimizer = 'adam',
              loss = 'binary_crossentropy',
              metrics = ['accuracy'])
```

We are using ReLU() again in the first fully connected layer, and Sigmoid() in the second one because this is a binary classification problem. In the compiling stage, we are using the 'adam' optimizer, which is a generalization of the stochastic gradient descent (SGD) algorithm(see Appendix). This is widely used to train CNNs.

5.2.3 Result Analysis

Once the training is complete, we get a 0.8054 accuracy, and the output in the terminal is shown below: It is clear that the loss drops while the



```
Epoch 1/10
1250/1250 [==============================] - 79s 63ms/step - loss: 0.6347 - acc: 0.6247
Epoch 2/10
1250/1250 [==============================] - 85s 68ms/step - loss: 0.5540 - acc: 0.7175
Epoch 3/10
1250/1250 [==============================] - 81s 65ms/step - loss: 0.5066 - acc: 0.7511
Epoch 4/10
1250/1250 [==============================] - 87s 69ms/step - loss: 0.4778 - acc: 0.7696
Epoch 5/10
1250/1250 [==============================] - 80s 64ms/step - loss: 0.4478 - acc: 0.7858
Epoch 6/10
1250/1250 [==============================] - 85s 68ms/step - loss: 0.4247 - acc: 0.8054
Epoch 7/10
1250/1250 [==============================] - 81s 65ms/step - loss: 0.4007 - acc: 0.8141
Epoch 8/10
1250/1250 [==============================] - 82s 65ms/step - loss: 0.3835 - acc: 0.8241
Epoch 9/10
1250/1250 [==============================] - 85s 68ms/step - loss: 0.3635 - acc: 0.8371
Epoch 10/10
1250/1250 [==============================] - 81s 65ms/step - loss: 0.3395 - acc: 0.8486
```

Figure 5.8: Results from Ten Epochs

accuracy increases with each epoch. It is impressive that the CNN gains a relatively high accuracy through this training process with just a few lines of code.

5.2.4 Images Predictions Analysis

We can classify the images according to three categories:

- **Strongly right predictions**: the model predicted these images correctly, and the output value is > 0.8 or < 0.2
- **Strongly wrong predictions**: the model predicted these images wrongly, and the output value is > 0.8 or < 0.2
- **Weakly wrong predictions**: the model predicted these images wrongly, and the output value is between 0.4 and 0.6

Figure 5.9 shows nine randomly selected images from the strongly right group:

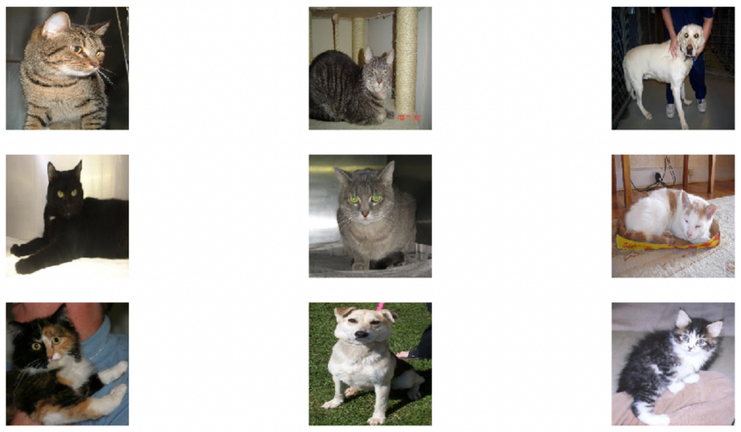


Figure 5.9: Selected images that have strong predictions and are correct

These are almost classic images of cats and dogs, and it seems that the CNN captures the unique features of each group correctly. Cats have pointy ears and dogs have black eyes, which allow our CNN to easily classify them.

Figure 5.10 shows nine randomly selected images from strongly wrong group:

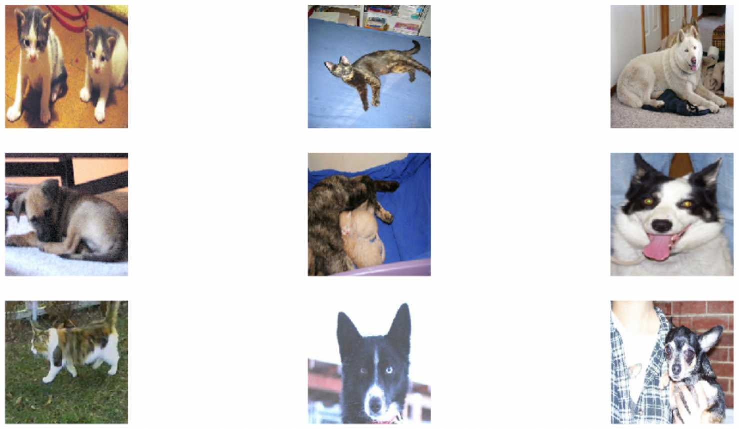


Figure 5.10: Selected images that have strong predictions but are wrong

A few commonalities exist among these strongly wrong predictions. Certain dogs do resemble cats with their pointy ears. Perhaps our CNN put too much emphasis on this ear feature and classified them as cats. Some objects are not even facing the camera, which makes the CNN unable to find certain characteristics or misunderstand some unrelated features.

In Figure 5.11, there are several vague pictures showing up. These images are difficult to identify at the first place even as human beings because some are very dark, and some involve other objects. Our CNN probably fails due to the missing shared features in these images.

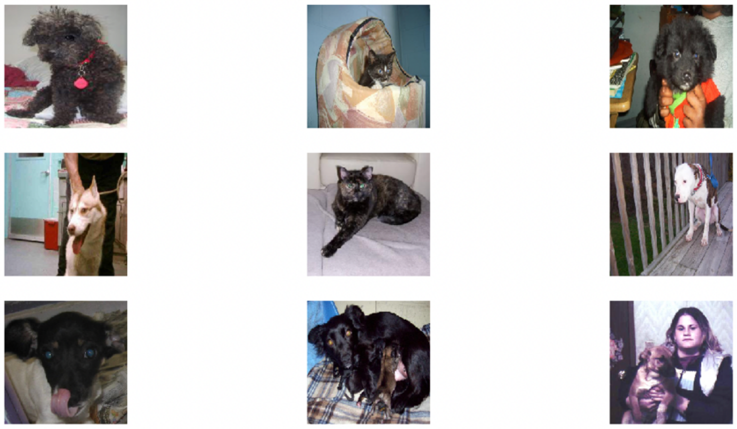


Figure 5.11: Selected images that have weak predictions, and are wrong

This model could be generalized to do more advanced classification problems. Although the accuracy is not perfect, it still provides a lot of convenience for us if we have awaiting images ready for classification.

Chapter 6

Further Topics and Potential Future

We have discussed many aspects of neural networks, and the main purpose of this 'human-like' computer program is to improve the efficiency when we solve problems with massive amount of data. People gain experience from the past over many years, while the neural networks learn from the past and predict for the future in a short time. It is also important to know their strengths and weaknesses, so we could know when to apply them and how they could be improved to further help us deal with problems.

6.1 Key Strength of Neural Networks

- Performance on problems with large data sets:
  There are numerous problems in the real world which require us to take multiple variables into account, and neural networks are good

at handling these types of problems. The algorithms are already set inside the neural networks, so that they are ready for massive amount of data, and more data bring in more accurate training and higher performance. In contrast, more data does not improve the performance of those traditional machine learning approaches.

- Feature Engineering:
  Neural networks are also good at capturing correct features, which is called feature engineering. The incredible part is that neural networks can figure out what are the important features which need to be taken into consideration, without any help or guidance from human beings.

- Applicability:
  One important property of neural networks is that they have the power of flexibility. Once established, they can be applied to almost anything, helping people to differentiate data into classes in a small amount of time. Therefore, if we have a neural network that can learn to recognize patterns, it could feasibly recognize patterns in almost any domain.

6.2 Key Weaknesses of Neural Networks

- Data requirements:
  At the beginning stage, all neural networks need to go though a learning period where they start to recognize patterns and refine themselves. However, the data selection is a main problem for starters. There is still a massive data requirement before those algorithms can start to be effective. We need to have enough time to include all the required data, otherwise the performance of the neural networks will be limited.

- Expensiveness:
  The neural networks require us to have enough memory space to store all the data and strong GPU and CPU beyond the scope of a normal system. Individuals may be unable to build their own neural network due to this limitation, and currently in practice, neural nets are widely used in those tech companies which can afford the expense of implementing neural networks.

- Opaqueness and blindness:
  Once we set up neural networks, it is hard for us to further develop them. Because the data are passed through a complicated system, from outside we do not know exactly what they encounter in the system. Additionally, it is not transparent about the performance of the built in algorithms. There is still a long way to go if we would like to know how we get the answers and how we can improve the performance of a neural network.

- Long-term potential:
  We know that a neural network is a more advanced machine learning program because its data processing ability is close to the automatic level. However, in practice the limitation is the hardware; we still have not had the ideal computer, which stores infinite amount of data with a high speed of processing time. The future of ANNs depends on the future of our technology, which still awaits unraveling.

6.3 What's in the future?

The ANN model is already applied to several fields right now. The voice system of Alexa, which is a smart-home device from Amazon and the direction feature of Google Maps are examples of applications. Neural networks, and the theory behind them are called deep learning. and they are currently changing our lives. They will even become more important in the future. They can be augmented with some operations borrowed from classical approaches to make predictions in an efficient way, so that people can benefit from this convenience by doing less work. Additionally, we may explore more about how to invert through a neural network to better understand its architecture, since right now we are facing a problem called 'Black-box', that is, we do not really know how to trace back from the results a neural network produces. Once we figure out how to trace back from the results, we can adjust algorithms accordingly to increase both their efficiency and their accuracy.

Bibliography

[1] Loy, James. NEURAL NETWORK PROJECTS WITH PYTHON: the Ultimate Guide to Using Python to Explore the True Power... of Neural Networks through Six Projects. PACKT PUBLISHING LIMITED, 2019

[2] Ujjwalkarn. “A Quick Introduction to Neural Networks.” The Data Science Blog, 10 Aug. 2016, https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/

[3] Woodford, Chris. “How Neural Networks Work - A Simple Introduction.” Explain That Stuff, 4 Apr. 2019, https://www.explainthatstuff.com/introduction-to-neural-networks.html

[4] Babs, Temi. “The Mathematics of Neural Networks.” Medium, Coinmonks, 14 July 2018, https://medium.com/coinmonks/the-mathematics-of-neural-network-60a112dd3e05

[5] Activation Functions, https://en.wikipedia.org/wiki/Activation_function

[6] Bhoge, Manish. “Using the Artificial Neural Network for Credit Risk Management.” DataScience.com, 23 Jan. 2019, www.datascience.com/blog/artificial-neural-network-credit-risk-management

[7] Min Li, Amy Mickel Stanley Taylor (2018) “Should This Loan be Approved or Denied?”: A Large Dataset with Class Assign-

ment Guidelines, Journal of Statistics Education, 26:1, 55-66, DOI: 10.1080/10691898.2018.1434342, https://doi.org/10.1080/10691898.2018.1434342

[8] "A Beginner's Guide to Convolutional Neural Networks (CNNs)." Skymind, skymind.ai/wiki/convolutional-network

[9] Nielsen, Michael. "How the Backpropagation Algorithm Works." Neural Networks and Deep Learning, Determination Press, Oct. 2018, neuralnetworksanddeeplearning.com/chap2.html

[10] Donges, Niklas. "Pros and Cons of Neural Networks." Towards Data Science, Towards Data Science, 17 Apr. 2018, towardsdatascience.com/hype-disadvantages-of-neural-networks-6af04904ba5b.

Chapter 7

Appendix

7.1 Widrow-Hoff Rule

This is introduced by Bernard Widrow and Marcian Hoff. It is also called Least Mean Square(LMS) method, to minimize the error over all training patterns.

This rule is based on the gradient-descent approach, which continues forever. It updates the weights by the following formula:

$$\Delta w_i = \alpha x_i e_i$$

where Δw_i is the weight change for ith pattern, α is the learning rate, x_i is the input, and e_i is the error(i.e. the difference between the actual value and the desired value).Note that the above rule is for a single output only, and the weight is updated in either of the two cases below:

Case 1: $t \neq y$, then

$$w_{new} = w_{old} + \Delta w$$

Case 2: $t = y$, then no change in weight.

7.2 Clustering Algorithm: k-means

When we have a training set, we have already known the classifications happened in the past, which are extremely imbalanced results. We would like to pre-process the dataset in order to eliminate the imbalance, so we use the clustering algorithm k-means to cluster the majority category into k subgroups, and merge the k subgroups of the majority category data and minority category data respectively into k balanced subgroups in order to have a diverse set. A visualization of k-means algorithm is shown in Figure 7.1:

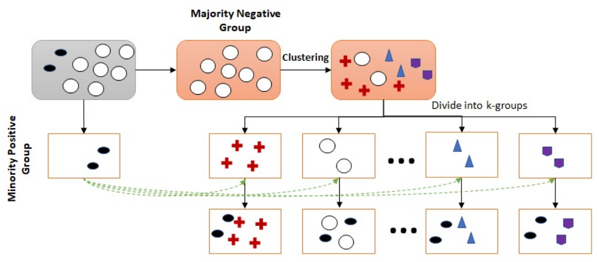


Figure 7.1: K-means Algorithm to Pre-process Data

K-means clustering is an unsupervised approach, which is used when we have unlabeled data(i.e., data without defined groups). This algorithm starts with k classes, which are pre-determined by us. Given k center points, we call them

$$c_1, c_2, c_3, ..., c_k$$

and we have data points

$$x_1, x_2, x_3, ..., x_j$$

We have three steps to find the clusters:

1. Sort the data: for a certain data point x,if $||x - c_i|| \leq ||x - c_k||$ for all k, then x belongs to cluster i.
2. Recompute the center i: given $x_1, ..., x_p$ in cluster i,
$$c_i = \frac{1}{p} \sum_{i=1}^{p} x_i$$
3. Repeat step 1 and step 2.

An important concept involved here is the Distribution Error: if $x_1, ..., x_p$ belong to cluster i,
$$Err = \frac{1}{p} \sum_{n=1}^{\infty} ||x - c_i||$$
The iteration of this algorithm stops when the distribution error does not decrease anymore.

7.2.1 Choosing K

The algorithm k-means finds clusters for a particular pre-chosen k. In order to find this number, we need to run this algorithm for a range of k values and compare the results. What we will get is an accurate estimate, because there is no method to determine the exact value of k.

One commonly used way to find k is using the mean distance between data points and their clustered centroid to compare results across different values of k. Mean distance to the centroid as a function of k is plotted and the "elbow point," where the rate of decrease sharply shifts, can be used to roughly determine k. This approach is similar to that of avoiding over-fitting problem in neural networks.

Figure 7.2 is an example of elbow point. We can see that the performance stops to changes at $k = 4$.

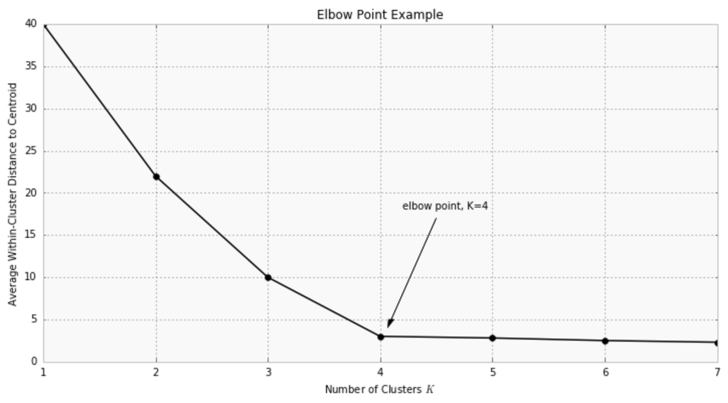


Figure 7.2: How to find the value for k?

Additional information on k-means could be found here:`https://www.datascience.com/blog/k-means-clustering`