國立政治大學應用數學系

碩士學位論文

BERT 應用於數據型資料預測之研究：以美國職棒大聯盟全壘打數預測為例

Using BERT on Prediction Problems with Numeric Input Data: the Case of Major League Baseball Home Run Prediction

指導教授：蔡炎龍　博士

研究生：孫瑄正　撰

中 華 民 國　109　年　6　月

DOI:10.6814/NCCU202000709

致謝

兩年的碩士生涯過得好快，沒想到有一天就這樣畢業了。

從會計系過來之後才發現這個世界好大好大，數學學了更多之後才越來越發現自己的不足，必須時時刻刻告訴自己需要不斷努力，有扎實的基礎之後，才能夠繼續往下走。

謝謝炎龍老師，當初沒有老師的推坑還有不斷的讓我問問題，就沒有今天有能力口試的我，非常謝謝老師的指導，也謝謝老師在很多方面都願意相信我，讓我自由地做想做的事；謝謝澤佑，從我是一個什麼都不懂的大二，講到讓我明白不少事情的碩二，一路上也很謝謝你的建議，不管數學、程式或是人生；很謝謝大哥，我的咖啡跟你學了不少，還吃了你很多的晚餐還有甜點；很謝謝老大，不僅用數學豐富了我的碩士生涯，同時也用綠色瓶子的飲料點綴了我的人生；謝謝瑞鋆，每個禮拜都從新竹上來教我微分幾何，希望你的博士生涯一路順遂；也很謝謝研究生室的大家，在這邊互相扶持也一起歡笑，所有能聚在一起的時刻都非常棒，是我在碩班非常珍惜的一件事，也是政大最好的研究生室；謝謝我的家人，在我做了去唸數學的決定時仍然支持我；謝謝女友，在這段過程中的陪伴，使我有力氣繼續往目標邁進。

謝謝所有遇到的人事物，這些都是我在碩士生涯中，得到最珍貴的回憶。

DOI:10.6814/NCCU202000709

中文摘要

BERT 在自然語言處理的領域中是一個強而有力的深度學習的模型，它的模型架構使得它可以透徹的了解我們使用的語言，在不同的任務中像是機器翻譯或是問答任務上都有很不錯的成果。在本篇論文中，我們證實了 BERT 可以使用數據形態的資料去預測結果，並且實際上做了一個例子，探討它在數據型資料輸入時的表現，我們將美國職棒大聯盟球員的數據作為輸入，使用 BERT 進行關於球員未來全壘打表現的預測，並且將其預測結果與 LSTM 以及現行球員表現預測系統 ZiPS 做比較。我們發現在 2018 年的測試資料中，使用 BERT 預測的準確率高達 50%，LSTM 有 48.8% 而 ZiPS 只有 25.4%；在 2019 年的測試資料中，雖然表現略有下滑，但 BERT 的 44.4% 準確率仍舊高於 LSTM 的 42.8% 以及 ZiPS 的 30.1%。總體來說，BERT 能夠對於數據形態的資料有深度的了解，使得它的表現比起傳統的方式來說更加穩定和精確，同時我們也找到了球員表現預測的一個新方法。

關鍵字：BERT、棒球、深度學習、長短期記憶模型、神經網路、球員表現預測、預測系統、Transformer

DOI:10.6814/NCCU202000709

Abstract

BERT is a powerful deep learning model in nature language processing. It performs well in various language tasks such as machine translation and question answering since it has great ability to analyze word sequence. In this paper, we show that BERT is able to make prediction with numerical data input instead of text. We want to predict output with numerical data and verify its performance. In particular, we choose the home run performance prediction task which input the stats of players in Major League Baseball. We also compare result of BERT-based approach with the performance of LSTM-based model and the popular projection system ZiPS. In testing data of year 2018, Bert-based approach reaches 50.6% accuracy while LSTM-based model has 48.8% and ZiPS gets only 25.4% accuracy rate. In 2019, BERT achieves 44.4% accuracy but 42.8% of LSTM-based and 30.1% of ZiPS. BERT is not only able to handle the numerical input with time series, but also performs stably and better than those traditional methods. Moreover, we found a new effective way in player performance prediction.

***Keywords*—** BERT, baseball, deep learning, long short-term memory, neural network, player performance prediction, projection system, Transformer

DOI:10.6814/NCCU202000709

Contents

DOI:10.6814/NCCU202000709

DOI:10.6814/NCCU202000709

List of Tables

DOI:10.6814/NCCU202000709

List of Figures

DOI:10.6814/NCCU202000709

Chapter 1

Introduction

In recent years, deep learning approaches achieve unprecedented performance on a broad range of problems from various area. Sequential deep learning model such as Recurrent Neural Networks (RNN) [31] and Long Short-Term Memory (LSTM) [13] have proven very powerful for applications in the data with time series. In 2017, Google announced a whole new model called "Transformer" [37] which based on attention and has impressive performance in Natural Language Processing (NLP) tasks. Bidirectional Encoder Representations from Transformers (BERT) [7] is a modified model from Transformer. It reached state-of-the-art in many tasks such as neural machine translation and question answering problem. BERT has ability to get information about sequence input with its good embedding. Since BERT is able to predict language sequence data so well, we are curious about the input of numeric sequence data. We want to use the ability about BERT dealing with language to solve the problem with number sequence. It will be a new effective way in the prediction of numeric data with time series by the structure of BERT. Therefore, we aim to investigate the performance of BERT when we input the numerical data in this paper. In particular, we focus on the prediction of player performance in Major League Baseball (MLB).

Baseball is one of the most popular sports around the world. People watch the ball game and talk about players and teams they like after a busy day. There are several organized professional baseball leagues such as MLB and Minor League Baseball in the United States and Canada and Nippon Professional Baseball (NPB) in Japan. MLB is the most well-known baseball league around the world. Since professional baseball players are increasingly guaranteed expensive long-term contracts, team managers (such as team owner or coaches) tend to understand each

DOI:10.6814/NCCU202000709

player's status beforehand. Therefore, it is important to predict players' performance for the coming year. Then, team managers could figure out who have the potential to be the rising star, and who should be traded immediately. Therefore, performance prediction system is valuable in practical since it provides additional information for team managers to make a better decision. Most performance prediction systems are constructed based on historical data of each player. In other words, most prediction systems are constructed based on regression analysis or seasonal time series analysis. Hence, we also want to create a new prediction system by deep learning.

In this paper, we only predict individual baseball players' home runs (HR) in MLB because it is one of the most critical index to understand the power and the talent of a baseball player. We use BERT and propose several deep learning models based on the LSTM structure. To evaluate the capacity of our models, we compare our prediction results to other prediction systems used in practical.

Our code is available at https://github.com/hsuanchengsun/BERT-baseball.

DOI:10.6814/NCCU202000709

Chapter 2

Related Work

In this chapter, we talk about related work in two parts: numerical data applied on BERT and using neural network to predict baseball players' stats. However, there is less case about the first topic. We believe that there will be more and more research on this in the future.

On the other topic, Lyle (2007) had used artificial neural networks and other machine learning methods to predict six of hitter's stats [21]. He focused on ensemble learning and compared the results with the other prediction systems. The performance of his methods outperforms the prediction system in three items.

Koseler and Stephan (2017) mentioned that there is only 9% of baseball analysis research that neural network is one of their method [17]. They believed that the situation will be changed in the future since deep learning is an effective way in application.

DOI:10.6814/NCCU202000709

Chapter 3

Deep Learning

Artificial intelligence (AI) technology has been an important study since late of last century which aim to make computer acting like a human. We started from some simple model [24]. We multiplied input with weights to fit the real output. However, it is not easy to modify the weights in model with complex structure after we measured the difference between prediction and real answer. Back propagation announced in 1986 [31] had solved this problem and be the effective way to update parameters in a deep structure. Deep learning are not a robust method until database and hardware update. It shows the ability to handle huge amount of data with systematic model. In 2012, AlexNet [18], which based on convolution neural network (CNN) [19], became the champion of the Large Scale Visual Recognition Challenge (ILSVRC) which is the biggest competition of object detection and image classification in the world. AlphaGo [32] also beat professional go player by the power of deep learning. Nowadays, deep learning is used in variety of field, such as object detection [29], natural language processing (NLP) [28] and image classification [11].

Generally, deep learning is a structure of artificial neural network (ANN) [15] with at least one layer. Before taking a close look at deep learning structure and how does it work, we need to know about the concept of neural network. The basic idea is to make computer be able to simulate what human think. After seeing a picture or listening to people, we will judge these thing and know the exact meaning of them. That process just like a function. Deep learning is trying to simulate the pattern from our brain to the computer. We aim to develop a function f to copy all the things, for example,

$$f(\text{“I love you.”}) = \text{“I love you, too.”}$$

DOI:10.6814/NCCU202000709

No matter the input is a picture or a sentence, we can all have a suitable output. But what is the system in our brain? Well, we got an answer from the researches in biology [20]. Neuroscientists found that there is a complex connection between different neurons in our bodies. The huge neural network help human to think and life. Hence, we would like to copy the structure on the computer and that is ANN. We hope deep learning is the function such that computers have the same activities as human. By the good device like graphic processing unit (GPU) and ability to build larger datasets, our result gets better day after day.

3.1 Neuron and Neural Networks

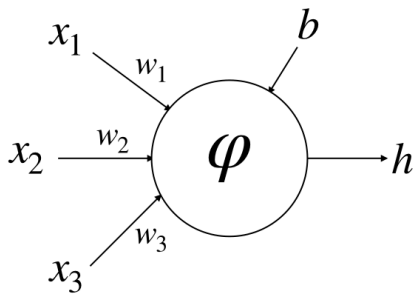

Figure 3.1: A neuron in neural netwoks

As shown in Figure 3.1, suppose x_1, x_2 and x_3 be the input of the neuron in $\mathbb{R}$. Then we will give three random independent weights w_1, w_2 and w_3 to each of them, respectively, which would be adjusted while training the model. There is also a flexible bias b in the neuron, so we have the operation $w_1x_1 + w_2x_2 + w_3x_3 + b$ now. Finally, an activation function φ will make the output h for the neuron i.e.

$$h = \varphi(\sum_{i=1}^{3} x_i w_i + b)$$

A full picture of an example of neural network structure is showed in Figure 3.2. It has three parts for total: input layer, hidden layers and output layers in the left side, middle and

DOI:10.6814/NCCU202000709

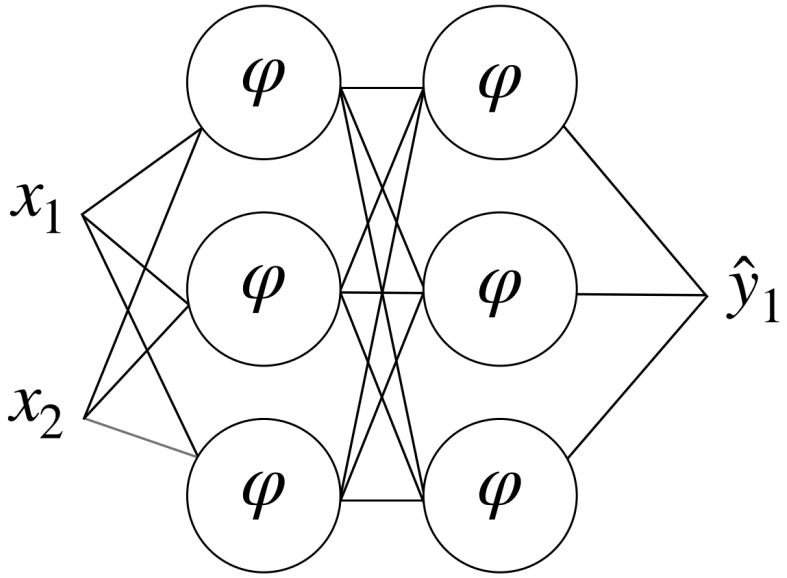

Figure 3.2: The structure of neural netwoks

right side, respectively. It presents that the output is predicted by the neurons in hidden layers after input data are given. We focus on how to use lots of neurons in hidden layers to get the outputs corresponding to the inputs. The framework of hidden layer can be complex or simple. Building such kind of deep learning model is not difficult. First, we have to decide the amount of layers and neurons. Secondly, we use input data doing supervised learning. Finally, we set a loss function to evaluate the prediction and update the parameters to get a better result. I will explain all the detail step by step in the following chapter.

3.2 Activation Function

Activation function takes an important role in whole process which make the output being nonlinear to simulate the neurons behavior of human. Moreover, it produces a better result to the data in real world since most of situations are not able to be predicted linearly. Here we show some common activations functions:

1. Sigmoid

   Equation:

$$\text{sigmoid} : \mathbb{R} \to (0, 1),\ \text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

DOI:10.6814/NCCU202000709

Graph:

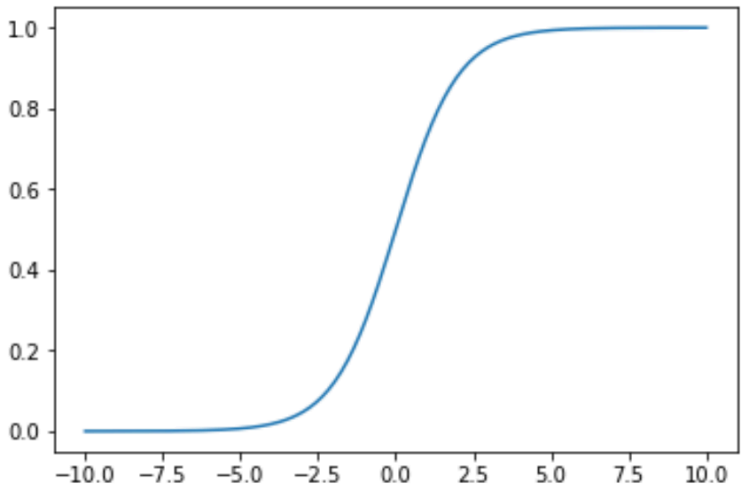

Figure 3.3: Sigmoid function

2. Hyperbolic tangent (tanh)

Equation:

$$\tanh : \mathbb{R} \to (-1, 1),\ \tanh(x) = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})}$$

Graph:

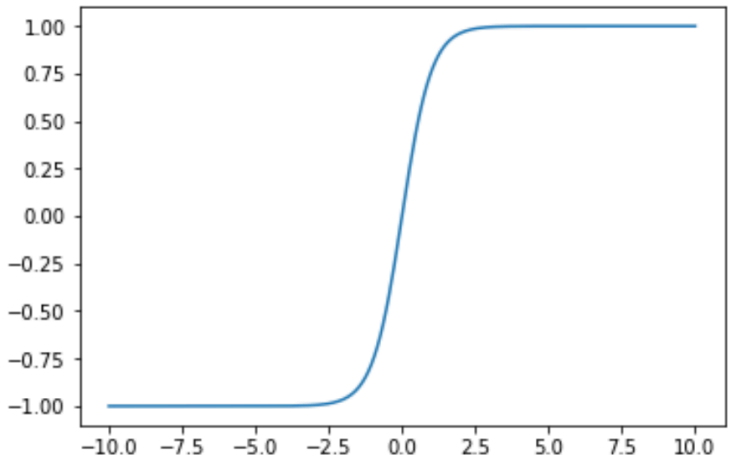

Figure 3.4: Hyperbolic tangent(tanh)

DOI:10.6814/NCCU202000709

3. Rectified Linear Units (ReLU) [25]

   Equation:

$$\text{relu} : \mathbb{R} \to [0, \infty), \ \text{relu}(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

   Graph:

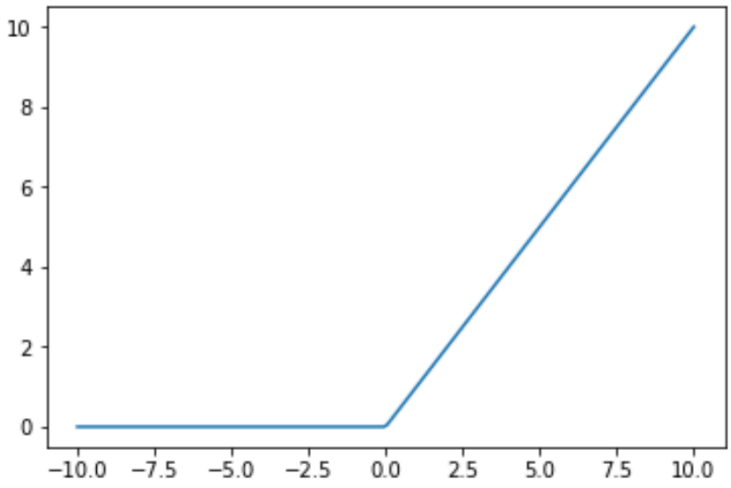

Figure 3.5: Rectified Linear Units(ReLU)

4. Softmax

   Equation:

$$\text{softmax} : \mathbb{R}^n \to \mathbb{R}^n, \ \text{softmax}(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

$$\text{for } i = 1, ..., \text{n where } x = (x_1, ..., x_n)$$

Of course, choosing linear function is fine. Generally, we choose the same activation function for every neurons in hidden layers and one for the final neuron depend on our output.

Now each neuron has its own output from the inputs. This process from the input layer to output layer through hidden layers is called feed-forward. Also, each output will be the input of all neurons in next layer, which connect whole networks. Hence, the structure shown in Figure 3.2 is also called fully connected feed-forward neural network.

DOI:10.6814/NCCU202000709

3.3 Loss Function

What should we do next? Remember that keeping the computer fitting the truth is our priority. After deciding the model structure, we have lots of parameters which equals to weights and bias, denoted by

$$\theta = \{w_i, b_j\},\ 1 \leq i \leq m, 1 \leq j \leq n,\ m, n \in \mathbb{N}$$

Then we do the addition and multiplication on them and get output after activation function. Denoted by f_θ, the final prediction of the neural network under the parameter set θ . Therefore, finding the best set of parameters θ^* to make the forecasting f_{θ^*} as close as possible to truth is our goal. But first of all, we need to define what is "close" to the answer y and our output $\hat{y}$. Here we need loss function L being the judge to tell us the quantization of distance between them. The exact number of $L(\theta) : \mathbb{R}^{m+n} \to \mathbb{R}$ mention the loss between the output under parameter set θ and the answer. Hence $L(\theta^*)$ will be our target which is the minimum number among all sets. We show some common loss functions below:

1. Mean Square Error (MSE)

   Equation:

   $$\text{MSE}(\theta) \ = \ \frac{1}{k} \sum_{i=1}^{k} \|y_i - f_\theta(x_i)\|^2$$

   where k is the total number of data.

2. Mean Absolutely Error (MAE)

   Equation:

   $$\text{MAE}(\theta) \ = \ \frac{1}{k} \sum_{i=1}^{k} |y_i - f_\theta(x_i)|$$

   where k is the total number of data.

3. Binary Cross Entropy (BCE)

   Equation:

   $$\text{BCE}(\theta) \ = \ \frac{1}{k} \sum_{i=1}^{k} [y_i(\log(f_\theta(x_i))) \ + \ (1 - y_i)(1 - \log(f_\theta(x_i)))]$$

   where k is the total number of data and each $y_i \ = \ 0$ or 1.

DOI:10.6814/NCCU202000709

4. Categorical Cross Entropy (CCE)

   Equation:

$$\text{CCE}(\theta) = \frac{1}{k}\sum_{j=1}^{c}\sum_{i=1}^{k} y_{ji}(\log(f_\theta(x_{ji})))$$

where k is the total number of data and c is the total categories of answers.

3.4 Gradient Decent and Backpropagation

After having $L(\theta)$, we need to adapt whole parameters to lower the loss. The way we used is called gradient decent. Here is an example for only one parameter in θ, see Figure 3.6.

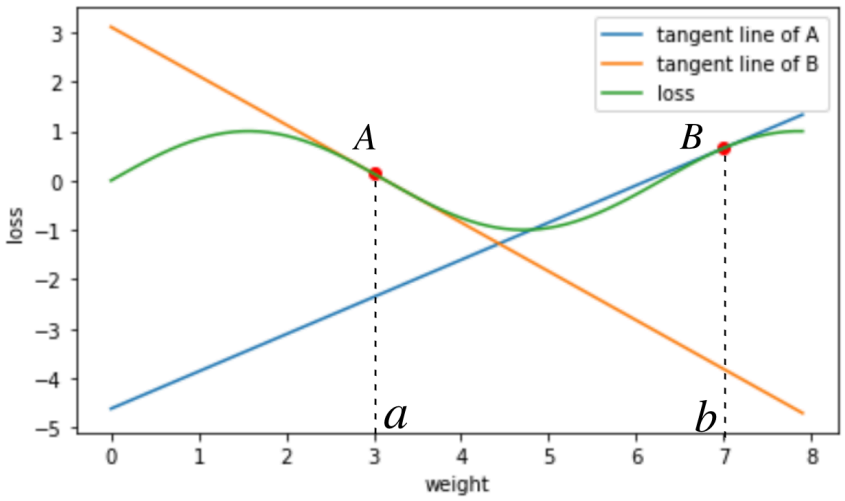

Figure 3.6: Gradient decent with one parameter

Suppose loss function L is the green line. X axis is the weight θ and Y axis denote the $L(\theta)$. Our goal is to get the minimum value of $L(\theta)$. Obviously, there are two points $A(a, L(a))$ and $B(b, L(b))$ located at the right and left side of the minimum location of L, respectively. We hope to move a and b to the minimal spot on loss function. Since the slope $L'(a) > 0$ at A, $a - L'(a)$ will be left of A which has a chance to get a smaller loss. Similarly, we can shift B right to reduce the loss by $b - L'(b)$ because $L'(b) < 0$. Hence, we obtain a conclusion quickly that we can have θ with lower loss by deducting the derivative for the parameter. This is the core concept of gradient decent. If we have several parameters, we can calculate the gradient i.e. partial derivatives of each weight and bias. The following shows the process:

DOI:10.6814/NCCU202000709

Suppose we have parameters set $\theta = \{w_1, w_2, ..., w_m, b_1, b_2, ..., b_n\}$ and a loss function L. We also give learning rate $\eta \in \mathbb{R}$ to control the speed of gradient decent. If the learning rate is too big, then we may not able to reach the minimum loss. On the other side, we will waste a lot of time if learning rate is too small. Hence, deciding a suitable learning for deep learning is important. For general siuation, we start the learning rate from a small value. The gradient ∇L and the new θ will be as following:

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial w_2} \\ \vdots \\ \frac{\partial L}{\partial w_m} \\ \frac{\partial L}{\partial b_1} \\ \frac{\partial L}{\partial b_2} \\ \vdots \\ \frac{\partial L}{\partial b_n} \end{bmatrix}, \theta^{new} = \begin{bmatrix} w_1^{new} \\ w_2^{new} \\ \vdots \\ w_m^{new} \\ b_1^{new} \\ b_2^{new} \\ \vdots \\ b_n^{new} \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \\ b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} - \eta \nabla L = \begin{bmatrix} w_1 - \eta \frac{\partial L}{\partial w_1} \\ w_2 - \eta \frac{\partial L}{\partial w_2} \\ \vdots \\ w_m - \eta \frac{\partial L}{\partial w_m} \\ b_1 - \eta \frac{\partial L}{\partial b_1} \\ b_2 - \eta \frac{\partial L}{\partial b_2} \\ \vdots \\ b_n - \eta \frac{\partial L}{\partial b_n} \end{bmatrix}$$

We usually use optimizer to adjust learning rate to help gradient descent finding the minimum value faster. There are two kind of method, one is changing through time, the other changes learning rate depend on the gradient [30]. Moreover, that is an efficient way to start the gradient decent from the final layer to the beginning. We can use chain rule to calculate the gradient of former layers easily. This procedure is called backpropagation [9] [31], which make the whole parameters be well adjusted.

3.5 Overfitting, Dropout and Batch Normalization

Training model is like teaching a student. We want this student have ability to solve the future problem by learning from training input. However, the student might be too smart. Our model can know everything about training data but it has no idea about how to predict testing stats. Overfitting is called to describe the phenomenon. It can be observed by performing

DOI:10.6814/NCCU202000709

perfectly during training with high accuracy and low loss but the prediction of testing data reflects low accuracy and high loss. Figure 3.7 shows the case of overfitting. Suppose orange line is the true function. We can have a great result to classify all the data with this line and it is the target of our model. However, our model learned the function as blue line which pass all the training data points. Clearly, it cannot predict testing data point very well. Overfitting is caused by a lot of situations. For example, lack of training data lead models to memorize them all. The complicated architecture of model may generate this problem [40]. Hence, we have to find some way to deal with it. These methods are called regularization.

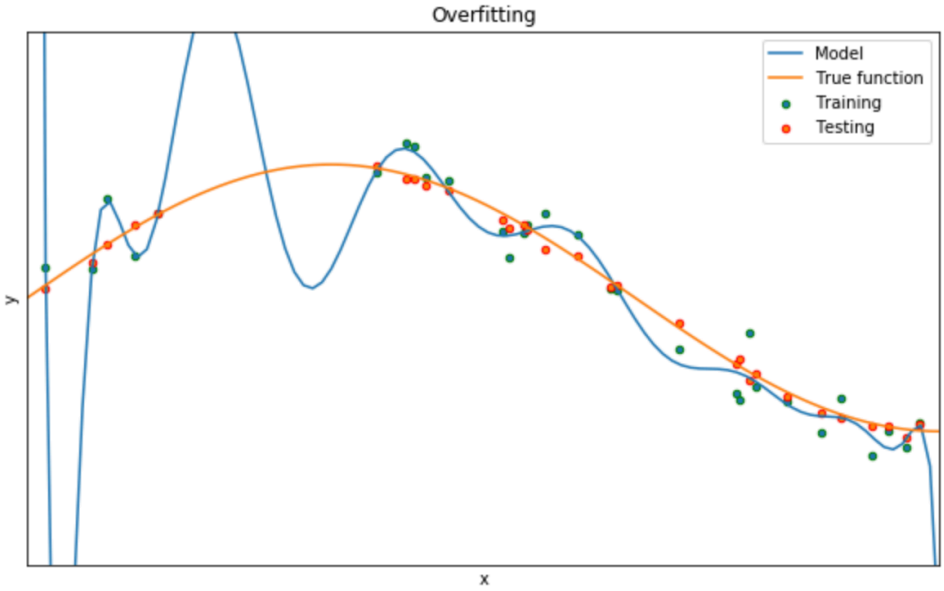

Figure 3.7: Overfitting

Traditionally, we add 1-norm(L1) and 2-norm(L2) of parameters to loss function as the penalty to let them being small [26]. L1-regularlizaion and L2-regularlizaion force the model not to weight too much on some features in training data which prevent overfitting. Dropout [12] is another simple way to avoid overfitting. It is similar to L1- and L2-regularlizaion which make model not to rely on some special features and it has less calculation then them. We are used to send a batch of data once a time instead of one data when we train the model. During each batch, we will randomly choose some outputs in hidden layer to be the input of next layer instead of all. Parameters in this thinned sub-network will be update by back-propagation while other masked

DOI:10.6814/NCCU202000709

ones stay the same. The process will be applied repeatedly which can be write as following:

$$
\begin{aligned}
r_j^l &\sim \text{Bernoulli}(p) \\
\hat{\mathbf{h}}^l &= \mathbf{r} * \mathbf{h}^l \\
h_i^{(l+1)} &= \varphi(\sum_{k=1}^{N} \hat{h}_k^l w_k^{l+1} + b_i^{(l+1)})
\end{aligned}
$$

where r_j^l denote the j-th random variable produced by Bernoulli distribution in the l-th hidden layer with probability p, $1 \leq l \leq L$ and $1 \leq j \leq N$. $\mathbf{h}^l$ is the output of l-th hidden layer and the k-th element denotes as $\hat{h}_k^l$. $*$ is the element-wise product. w_k^{l+1} and $b_i^{(l+1)}$ denotes the weights and bias in the i-th neuron in $(l+1)$-th hidden layer. φ denotes the activation function. When predicting with the model, the masked information is not feasible. Hence, we use dropout by multiplying each output of neuron in hidden layers by p to get the same expected value in training [34].

Batch normalization (BN) is the other regularization way though it is designed to solve the convergence of deep learning model [14]. For a simple way, we take the hidden layer as matrix computation. Suppose we have n input batch $\{\mathbf{x}_1, \cdots, \mathbf{x}_n\}$ with dimension d during training. Let X be the iuput matrix with i-th column vector be x_i, W be the weights matrix and B be the bias matrix, $1 \leq i \leq n$. Then the BN will be:

$$
\begin{aligned}
Z &= W \cdot X + B \\
\mu &= \frac{1}{n} \sum_{i=1}^{n} \mathbf{z}_i \\
\sigma^2 &= \frac{1}{n} \sum_{i=1}^{n} (\mathbf{z}_i - \mu)^2 \\
\hat{\mathbf{z}}_i &= \frac{\mathbf{z}_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \\
\hat{\mathbf{z}}_i' &= \gamma * \hat{\mathbf{z}}_i + \beta
\end{aligned}
$$

where $\mathbf{z}_i$ is the i-th column of Z, $\epsilon > 0$ to avoid $\sigma = 0$, γ and β are the parameter to be learned, and $*$ denotes the element-wise product. The final normalized output $\hat{\mathbf{z}}_i'$ can be applied activation function to be the input of next layer. While testing, we do not have mean and variance. We use global mean and variance from the whole data or use moving average and variance in training

DOI:10.6814/NCCU202000709

process. Our model can be more stable, converge faster if we operate BN. Also, the demand of regularization reduces.

DOI:10.6814/NCCU202000709

Chapter 4

Recurrent Neural Networks

An old school pun says that “Why shouldn’t we believe a man in bed?” The answer is that “Because he is lying.” For people know English, this is a common joke which the word has two meaning. However, it is a tough problem for computer to understand such complex thing with order. As mention in last chapter, deep learning is just like a function. Our input can be numbers, pictures and also sentence. How can neural network have ability to deal with text or data with time series? Well, recurrent neural network (RNN) [31] is an useful method to handle the problem. A RNN model get information from every past time input and has a strong ability to obtain the knowledge behind the squence. Hence RNN can make a good prediction about sequential data. It has a wide variety of application, such as knowing tomorrow’s weather by information of past n days, output a sentence from the previous one and generating a paragraph from a topic input. I will show all the detail in the following sections.

4.1 RNN Cell

Different form the neuron in normal neural networks, we call it a “RNN cell” as a basic element in RNN, see Figure 4.1. It still contains inputs, weights, output and activation function. But we have to take data series into account now. Suppose $\mathbf{x} = \{\mathbf{x}^t\}_{t=1}^{T}$ be a sequential data, $\mathbf{x}^t \in \mathbb{R}^n$ for all $1 \leq t \leq T, n \in \mathbb{N}$. Let $\mathbf{h}^t$ be the hidden state at time t, $\mathbf{h}^t \in \mathbb{R}^m$ for all $0 \leq t \leq T$, $m \in \mathbb{N}$, and $\mathbf{h}^0 = 0$. At each time t, we input the corresponding data $\mathbf{x}^t$ and the hidden state $\mathbf{h}^{t-1}$ which produced by the cell at time $t-1$. Hidden state will store the knowledge of input before time t and keep updating itself for each input after t. Weights for the data input and state

DOI:10.6814/NCCU202000709

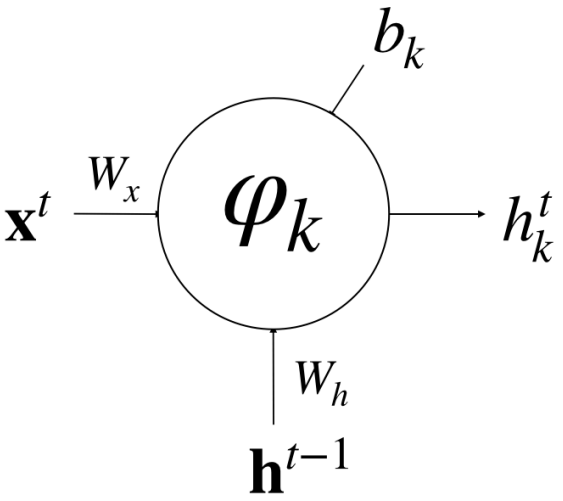

Figure 4.1: The structure of an RNN cell

and bias would be the same through time. General equation for the k-th cell status at time t can be written as:

$$h_k^t = \varphi_k(\mathbf{x}^t, \mathbf{h}^{t-1}) = \varphi_k(W_x^T \cdot \mathbf{x}^t + W_h^T \cdot \mathbf{h}^{t-1} + b_k).$$

where n is the dimension of data and m is the dimension of hidden state. $W_x \in \mathbb{R}^{n\times 1}$ is the data input weights metric and $W_h \in \mathbb{R}^{m\times 1}$ is the weights metric of hidden state input. b_k denotes the bias in φ_k.

Hidden state has not only the result of the cell, but also the information from the other cells in the same layer. Here is an easy example in Figure 4.2. For simply, weights and bias are not shown in the picture.

Suppose we have two cell φ_1 and φ_2 in a layer. $\mathbf{x} = \{\mathbf{x}^t\}_{t=1}^T$ is a sequential data in $\mathbb{R}^3$. At each time t, both cells will get input from the original data $x_1^t,\ x_2^t,\ x_3^t$ and the hidden state $h_1^{t-1},\ h_2^{t-1},\ 1 \le t \le T$. The output h_1^t from φ_1 will be:

$$h_1^t = \varphi_1(\sum_{i=1}^{3} w_{xi} \cdot x_i^t + \sum_{i=1}^{2} w_{hi} \cdot h_i^{t-1} + b_1)$$

Similarly, the output h_2^t from φ_2 will be:

$$h_2^t = \varphi_2(\sum_{i=1}^{3} w_{xi} \cdot x_i^t + \sum_{i=1}^{2} w_{hi} \cdot h_i^{t-1} + b_2)$$

DOI:10.6814/NCCU202000709

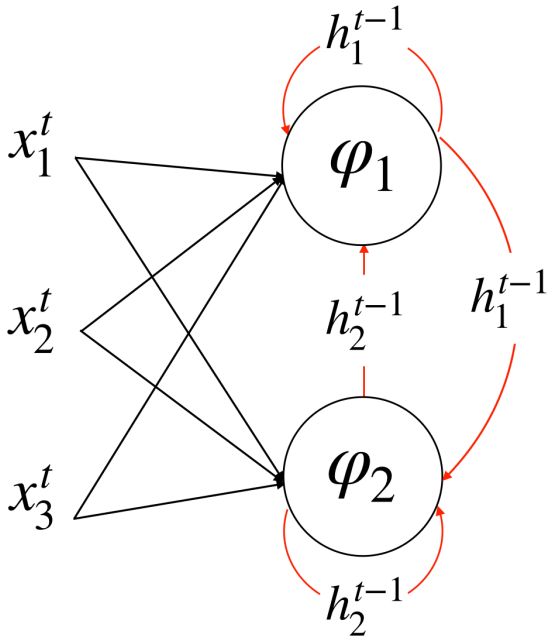

Figure 4.2: Multiple cells in one layer

Actually, hidden state can be the input for activation functions to generate output depend on our tasks. In the language model, we can see every word it produced. Or we just need the final result for weather prediction.

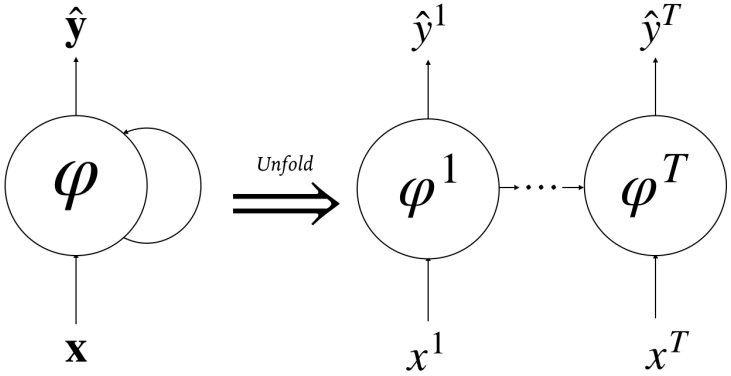

Figure 4.3: Unfolded RNN cell

As the structure on the left side in Figure 4.3, we use one RNN cell in the model. $\mathbf{y} \in \mathbb{R}^T$ and $\mathbf{x} \in \mathbb{R}^T$ are the prediction and input vector, respectively. We can also unfold the operation in RNN cell as the right side in Figure 4.3. It is similar to the normal NN but all the process from time 1 to time T take place in one cell.

DOI:10.6814/NCCU202000709

Then we can define the layer $\varphi : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m$ by

$$\varphi(\mathbf{x}^t, \mathbf{h}^{t-1}) = \varphi(W_x^T \cdot \mathbf{x}^t + W_h^T \cdot \mathbf{h}^{t-1} + b)$$

where n is the dimension of data and m is the dimension of hidden state, $n,\ m \in \mathbb{N}$. $\mathbf{x}^t$ and $\mathbf{h}^{t-1}$ are same as above. $W_x \in \mathbb{R}^{n \times m}$ is the data input weights metric and $W_h \in \mathbb{R}^{m \times m}$ is the weights metric of cell status input. $b \in \mathbb{R}^m$ denotes the bias. Moreover, let f be whole model which have multiple layers. If we collect all the hidden state at time t, we can write the prediction at time by

$$\mathbf{y}^t = f(\mathbf{x}^t, \mathbf{h}^{t-1})$$

The architecture of RNN is similar to NN, only the calculation of cell is differ from neuron. However, the structure of RNN in Figure 4.3 would make the model going too deep. This led to gradient vanishing [10] which is the reason of overfitting while training the model. To solve the unstable problem, we need a powerful model.

4.2 Long Short-Term Memory

The architecture of Long Short-Term Memory (LSTM) [13] which enhance the performance as well as deal with the unstable problem of original RNN [10]. Besides the initial cell unit, there are three more "gate" in a LSTM cell which be coefficients in $(0, 1)$ to adjust the input. Moreover, it outputs not only a cell status but also a memory from this cell. These improvements help LSTM know the information have to be forgotten, others need to be remembered.

At first we introduce gate in LSTM, as shown in Figure 4.4. Similarly to RNN cell, LSTM cell follows the recursive way. At time t, we have:

$$\begin{aligned} f^t &= \sigma(W_{xf} \cdot \mathbf{x}^t + W_{hf} \cdot \mathbf{h}^{t-1} + b_f) \\ i^t &= \sigma(W_{xi} \cdot \mathbf{x}^t + W_{hi} \cdot \mathbf{h}^{t-1} + b_i) \\ o^t &= \sigma(W_{xo} \cdot \mathbf{x}^t + W_{ho} \cdot \mathbf{h}^{t-1} + b_o) \end{aligned}$$

where $f^t,\ i^t,\ o^t$ are called forget gate, input gate and output gate, respectively. $\mathbf{x}^t$ is the data input at time t, and $\mathbf{h}^{t-1}$ is the hidden state at time $t-1$. They are all independent neuron, so they have different weight metrics and bias. σ denotes sigmoid function. Finally, let f be all the layers in RNN, we can collect all input at time t. The t-th prediction $\hat{y}^t$ will be $\hat{y}^t = f(\mathbf{x}^t,\ \mathbf{h}^{t-1})$

DOI:10.6814/NCCU202000709

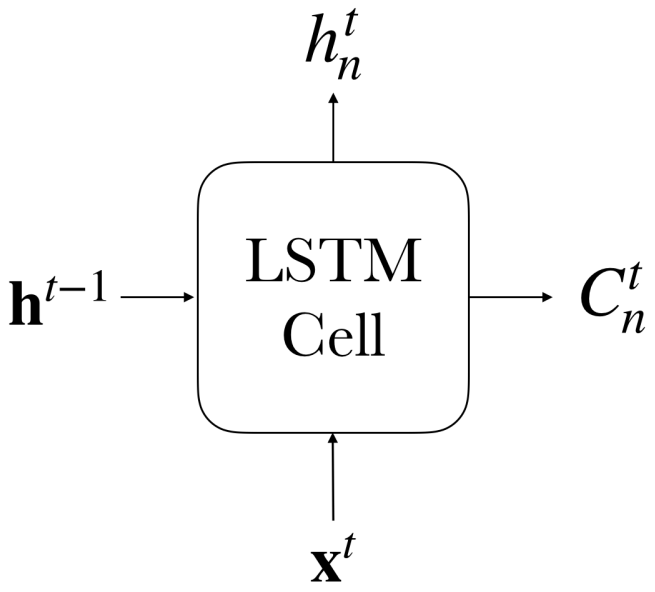

Figure 4.4: LSTM cell structure

Then these values will be used in the next step to produce cell state and hidden state:

$$\tilde{C}^t = \tanh(W_{xc} \cdot \mathbf{x}^t + W_{hc} \cdot \mathbf{h}^{t-1} + b_c)$$
$$C^t = f^t \cdot C^{t-1} + i^t \cdot \tilde{C}^t$$
$$\mathbf{h}^t = o^t \cdot \tanh(C^t)$$

where $\tilde{C}$ is another neuron in the cell. C^t is the cell state which stores information to build hidden state $\mathbf{h}^t$. It will not be input to other cells. Hidden state will share with other cells as usual. Generally, LSTM uses cell state to decide hidden state. It has three gates which are the best coefficients to adjust different part in LSTM. Teaching machine to know the importance of knowledge in different time is the purpose of LSTM.

4.3 Attention

Machine translation is one of the most important application in Natural Language Processing (NLP) projects. We need a sentence output after previous sentence has done. This kind of task called sequence to sequence(seq2seq) which is a classic problem in translation model. The unequal length of input and output sentence is the main difficulty. For example,

DOI:10.6814/NCCU202000709

『這是一本書』in Chinese means "This is a book" in English. Different length input makes RNN and LSTM cell had a hard time. In 2014, a research team from Google announced an encoder-decoder liked neural network which solved sequence to sequence problem perfectly [35]. They use two LSTM structure which modified the progress from RNN Encoder−Decoder [4] and had a better performance.

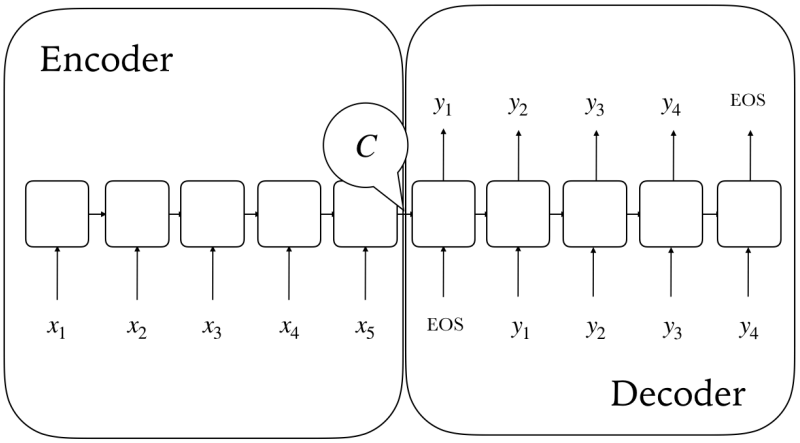

Figure 4.5: Encoder and decoder in `seq2seq`

Figure 4.5 is the structure of model. The words are the input. Squares in the middle is the LSTM cell in different time. "EOS" is a token represents "end of sequenc" which mentions model the sentence ends here. C is the final output from encoder. Each part has its own RNN framework with different input. Encoder is the first part of model which started from the beginning to the word before EOS in input. Containing as many as possible useful information about the input is the role of it. In encoder, the variable-length sequence will be encoded to a fixed-length vector. Words will be input one by one at each time t. We just need final hidden $C = \mathbf{h}^T$ of time T. We consider it as the summary about the input sequence and is helpful for decoder generates output. Hence, let f_e denotes the LSTM in encoder, we have

$$\mathbf{h}^t = f_e(\mathbf{h}^{t-1}, x^t)$$
$$C = \mathbf{h}^T = f_e(\mathbf{h}^{T-1}, \mathbf{x}^T)$$

which are same as the process we know in last section.

In the next step, decoder use different RNN model to generate variable-length sequence from fixed-length vector. It begins from the EOS token in input sentence and stops while

DOI:10.6814/NCCU202000709

predicting EOS in the output. To distinct different LSTM in encoder and decoder, we set f_d be the LSTM in decoder, $\mathbf{s}^t$ be the cell status and $\mathbf{y}^t$ be the input at time t. For the hidden state $\mathbf{s}^t$ at time t, the input for RNN not only include the previous word and cell status but also contain the summary c we got from encoder. Then the word is predicted by the previous words and C with softmax. Therefore, we have the equation of hidden state in decoder:

$$\mathbf{s}^t = f_b(\mathbf{s}^{t-1}, y^t, C)$$

This kind of encoder-decoder model helps advancing the performance in machine translation.

However, it is not efficient to use a static summary c from whole input sentence to predict one word. As our example, 「書本」in Chinese means "book" in English, so they should have strong connection. But other words may not be helpful to predict "book". Therefore, it is useful if we can take different weights of every words in input sequence. This skills is called attention, which the decoder will pay attention to those parts in sentence input for they need [2]. First, we consider bidirectional RNN as the encoder. The original direction of RNN is from time 1 to time T, so we obtain hidden state from $\mathbf{h}_s^1$ to $\mathbf{h}_s^T$. In some case we need to finish it one more time on the reverse side, which we get hidden state from $\mathbf{h}_r^T$ to $\mathbf{h}_r^1$. That's all for bidirectional RNN. Here we combine $\mathbf{h}_s^t$ and $\mathbf{h}_r^t$ together to be $\mathbf{h}^t$ as the hidden state in encoder. Second, we calculate the score e_j^t for each hidden state in encoder at time t by

$$e_j^t = a(\mathbf{s}^{t-1}, \mathbf{h}^j),\ 1 \leq j \leq T$$

where a is called aligned model, which is a number represents how close of your j-th input and t-th output. The score depends on the relationship of them. Then we find weights α_j^t of each $\mathbf{h}_b^t$ by

$$\alpha_j^t = \frac{exp(e_j^t)}{\sum_{i=1}^{T} exp(e_i^t)},\ 1 \leq j \leq T$$

which is softmax. Last, we obtain attention $\mathbf{c}^t$ and hidden state s^t in decoder at time t by

$$\mathbf{c}^t = \sum_{j=1}^{T} \alpha_j^t \mathbf{h}^j$$

$$\mathbf{s}^t = f_d(\mathbf{s}^{t-1}, \mathbf{y}^t, \mathbf{c}^t)$$

This way improves the performance of model in NLP tasks. But there is a question: Can we use attention without RNN structure? We will talk about this topic in next chapter.

DOI:10.6814/NCCU202000709

Chapter 5

Bidirectional Encoder Representations from Transformers

5.1 Word Embedding

In NLP model, our input are usually words, but computers can only accept numbers. How can we switch words into numerical data for neural networks? This is a tough problem in NLP field which is called word embedding. We try to develop strong methods transferring words to numbers which are able to represent our complicated language by numbers. Intuitively, we use one-hot encoding, which is a n-dimensional vector, to embed the word. For example, we can count appearance of each word in the data we use. Then we sort the number and let $[1,\ 0,\ 0,\ \cdots,\ 0]$ to be the embedding for the word which shows the most. $[0,\ 1,\ 0,\ \cdots,\ 0]$ for the second most and so on. They are all n-dimensional vector which depend on totally n independent words. In an easy way, we can just done word embedding for n-dimensional one-hot encoding randomly to every words. However, this kind of embedding is not useful. Imaging that we have 10 thousand of words in a text with millions of words. Our input will be too large and the embedding result may not efficient for other texts. Hence, there are lots of research to lower the dimension of word vector and try to embed by the relation between words, such as word2vec [22] [23], GloVe [27] and ELMO [28].

DOI:10.6814/NCCU202000709

5.2 Transformer

In 2017, research team from Google announced a whole new model in deep learning called "Transformer" [37]. It made a huge difference by using nothing about the structure of RNN or CNN. Transformer only include self attention skill in its encoder and decoder architecture. It reached the state of the art in machine translation task at that time and became a popular structure in NLP. Here we introduce the framework.

In the beginning, we talk about the input and output of transformer. Same as other machine translation task, the original sentence will be input of encoder and the translation will be input of decoder. While predicting, the output word at time t will be the input at time $t+1$ to just like we mention in `seq2seq` model. This property is called auto-regressive. However, that would lead to inefficiency during training because a wrong prediction will make the next incorrect one. The model will take a long time to learn. Hence, we train the model with teacher forcing [38]. In other words, we always input the correct answer no matter what is the output of model to reduce training time.

Transformer divides every words to sub-word units [39] which is called wordpiece embedding and switch them to index sequence. For example, "transformer" will be split to "trans" and "former" and the sequence will be $[1, 2]$. By this way, we can easily compose any word in the world. Then we will give them an index in the beginning and the other one in the end which represent "start of sequence" (<SOS>) and "end of sequence" (<EOS>). We will set a maximum number of sequence length and eliminate all the sequence which is longer than that in the next step. We also fill up every sequence to the maximum length with the index of "padding" (<pad>). Finally, each token will be embedded to d_{model} dimension. Of course we have different embedding way for different language. Transformer is also able to produce multiple words at the same time. Suppose input of encoder with dimension N and input of decoder with dimension M which from the first token to the one before last token. Combining all features above, the output dimension of transformer equals to M which beginning from the second token to the last one. Tokens can be decode to the wordpiece from the output.

Besides the word embedding, transformer add the information about position as well. Positional embedding will be summed with the word embedding, and be the final input. Each

DOI:10.6814/NCCU202000709

value of positional embedding will be

$$\text{PE}(pos, 2i) = \sin(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}})$$
$$\text{PE}(pos, 2i-1) = \cos(\frac{pos}{10000^{\frac{2i-1}{d_{\text{model}}}}})$$

for $1 \leq i \leq \frac{d_{\text{model}}}{2}$ and $1 \leq pos \leq d_{\text{model}}$. The combination of embedding not only make the model pay attention to the word but also the order. Moreover, they believe this kind of linear function help model know the relative position.

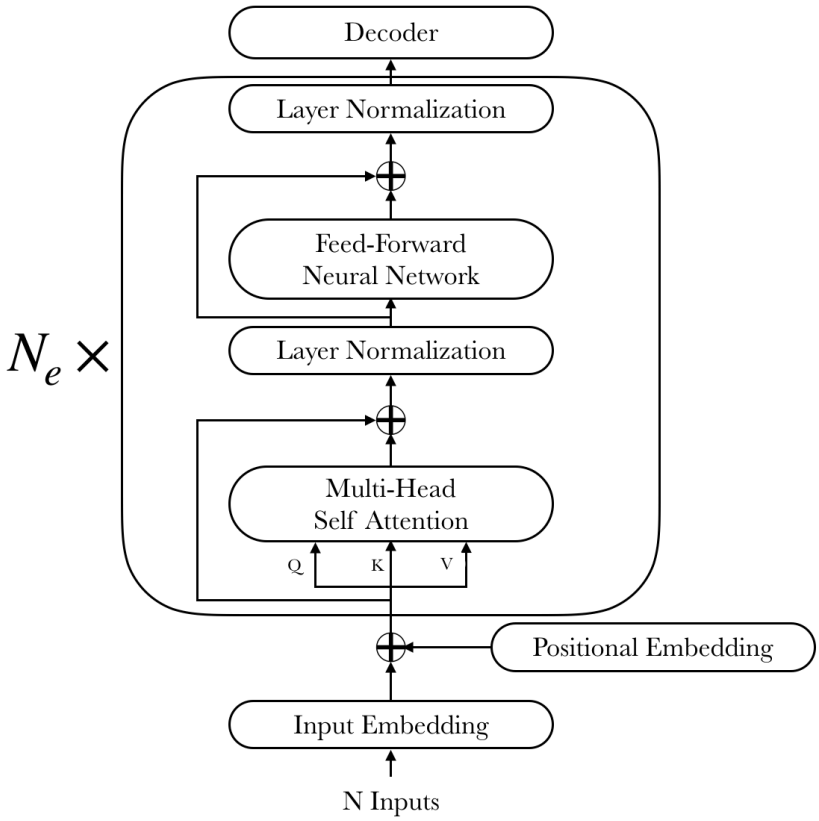

Figure 5.1: Encoder in Transformer

Next, we explain layer in the encoder. There are two sub-layers in it, first one is multi-head self attention and another is a fully connected feed-forward neural network (FFN) $\mathscr{F}_D$. There is also a residual connection [11] with layer normalization [1] following each sub-layers. The input vectors will be separated into three parts: key, value and query by multiplying distinct parameter matrices here. For example, let $x_1,\ x_2$ and x_3 be three input word vectors of dimension d_{model}.

DOI:10.6814/NCCU202000709

Then we get three query, key and value by:

$$q_i = W^Q \cdot x_i$$
$$k_i = W^K \cdot x_i$$
$$v_i = W^V \cdot x_i$$

for $1 \leq i \leq 3$ where W^Q and $W^K \in \mathbb{R}^{d_k \times d_{\text{model}}}$ and $W^V \in \mathbb{R}^{d_v \times d_{\text{model}}}$ denote the parameter matrices for three items where d_k and d_v are the dimension of key and value. Fixed q_1, we make the scaled dot-product attention $\alpha_{1,\,i}$ by

$$\alpha_{1,\,i} = \frac{k_i \cdot q_1}{\sqrt{d_k}} \text{ for } 1 \leq i \leq 3$$

to get the alignment for each key. These alignments will be applied softmax function to be the attention scores $\hat{\alpha}_{1,\,i}$ which are the coefficient corresponding to each v_i for $1 \leq i \leq 3$. Then we obtain output b_1 by the summation of these weighted values. Here are the equations:

$$\hat{\alpha}_{1,\,i} = \frac{exp(\alpha_{1,\,i})}{\sum_{j=1}^{3} exp(\alpha_{1,\,j})}$$
$$b_1 = \sum_{i=1}^{3} \hat{\alpha}_{1,\,i} \cdot v_i$$

Since summation of softmax function is 1, the output b_1 can be considered as a convex combination of v_i. The other outputs b_2 and b_3 are generated by the same process. The outputs comes from the information of itself instead of the way in RNN case. This kind of operation is called self attention. Clearly, we can complete the calculation of all input at the same time. Let $X \in \mathbb{R}^{d_{\text{model}} \times N}$ denotes N word vector of dimension d_{model} be the input. Hence, the matrix computation which can be shown as follows:

$$Q = W^Q \cdot X$$
$$K = W^K \cdot X$$
$$V = W^V \cdot X$$
$$\text{Attention}(Q,\ K,\ V) = [\text{softmax}(\frac{Q^T K}{\sqrt{d_k}} - 10^9 \cdot P)]^T V^T$$

where $W^Q \in \mathbb{R}^{d_k \times d_{\text{model}}}$, $W^K \in \mathbb{R}^{d_k \times d_{\text{model}}}$, $W^V \in \mathbb{R}^{d_v \times d_{\text{model}}}$ are parameter metrices for query,

DOI:10.6814/NCCU202000709

key and value, respectively. $Q,\ K,\ V$ are query, key and value which pack all $q_i,\ k_i,\ v_i$ together $\forall\ 1 \leq i \leq N$. d_k is the dimension of query and key. $P \in \mathbb{R}^{N \times N}$ is the padding mask matrix corresponding to the token sequence. The i-th column of matrix is 1 if the i-th token is <pad>, others are 0. We will multiply a huge negative number and add to $Q^T K$ such that the attention weights after softmax will be 0 to those padding position. The skill helps model skip those padding things. The output will be a weighted values vector of dimension d_v. The method is an effective way to speed up the training progress by GPU.

Only one is not enough, Google found that it is useful to execute several attention functions at the same time. $Q,\ K,\ V$ are the same as above. They will be weighted differently in each attention function. Each output is a head. Suppose we have h heads. We will concatenate all of them and multiply a parameter matrix to reshape the matrix to d_{model} dimension, which shows below:

$$\text{MultiHead}(Q,\ K,\ V) = W^O \cdot [\text{concatenate}(\text{head}_1,\ \text{head}_2, \cdots,\ \text{head}_h)]^T$$
$$\text{where head}_i = \text{Attention}(W_i^Q \cdot Q,\ W_i^K \cdot K,\ W_i^V \cdot V)$$

where the parameter matrices $W_i^Q \in \mathbb{R}^{d_k \times d_k}$, $W_i^K \in \mathbb{R}^{d_k \times d_k}$, $W_i^V \in \mathbb{R}^{d_v \times d_v}$ and $W^O \in \mathbb{R}^{d_{\text{model}} \times h d_v}$ for $1 \leq i \leq h$. concatenate is the function which concatenate all the matrices horizontally. The procedure helps model know sentence perfectly. Google concluded that each head has its own contribution to different tasks. Some heads will notice the syntactic structure and others learn the semantic construction.

Before moving forward to FFN, we firstly add the original input vector and output from multi-head attention and perform layer normalization $\text{LayerNorm}(X + \text{MultiHead}(Q,\ K,\ V))$ which can be show:

$$\mu_j = \frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} x_{ij}$$
$$\sigma_j = \sqrt{\frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} (x_{ij} - \mu_j)^2}$$
$$\hat{x}_{ij} = g_{ij} \frac{x_{ij} - \mu_j}{\sigma_j} + ij\ \forall 1 \leq i \leq N,\ 1 \leq j \leq d_{\text{model}}$$

where x_{ij} is the element in $X + \text{MultiHead}(Q,\ K,\ V)$ matrix, $\hat{x}_{ij}$ is the element in the matrix after

DOI:10.6814/NCCU202000709

performing layer normalization, g_{ij} and b_{ij} are parameters which was learned during training.

Next sub-layer is a position-wise fully connected feed-forward neural network with input and output dimension $= d_{\text{model}}$. There is only a hidden layer of dimension d_{ff} with ReLU in the middle. Each column vector $\hat{\mathbf{x}}_1, \hat{\mathbf{x}}_2, \cdots, \hat{\mathbf{x}}_N$ is the input and each position in vector shares the same parameters. Then $\mathscr{F}_D(\hat{\mathbf{x}}_i)$ can be write as following:

$$\mathscr{F}_D(\hat{\mathbf{x}}_i) = W^{'} \cdot \text{ReLU}(W \cdot \mathbf{x}_i + B) + B^{'}$$

$$\text{where ReLU}(W \cdot \mathbf{x}_i + B) = \max(0, \sum_{j=1}^{d_{\text{model}}} x_{nj} w_{nj} + b_n)$$

$\forall\ 1 \leq i \leq N,\ 1 \leq j \leq d_{\text{model}},\ 1 \leq n \leq d_{\text{ff}}$ where $W \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}},\ B \in \mathbb{R}^{d_{\text{ff}} \times 1},\ W^{'} \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}},\ B^{'} \in \mathbb{R}^{d_{\text{model}} \times 1}$ are parameters. $x_{nj},\ w_{nj}$ and b_n are denoted as the n-th element in $\mathbf{x}_j,\ W$ and B, respectively. Again, residual connection and layer normalization are performed after the fully connected neural network. Finally, we get output of the layer and will be input to next one. We have a stack of N_e layers in encoder. In original paper, $\text{N}_e = 6,\ d_{\text{model}} = 512,\ h = 8,\ d_v = d_k = 64$ and $d_{\text{ff}} = 2048$. The structure is shown in Figure 5.1.

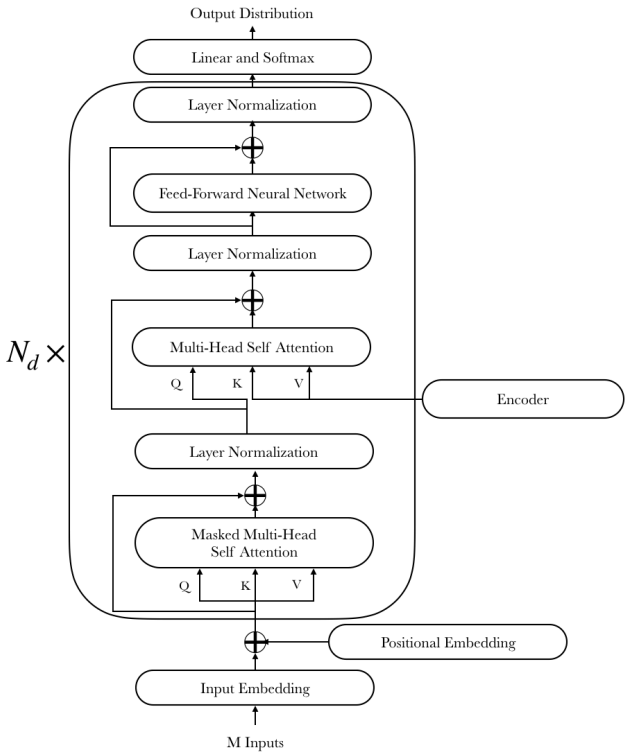

Figure 5.2: Decoder in Transformer

DOI:10.6814/NCCU202000709

When encoder has done, it's time for the decoder. Figure 5.2 is the picture of decoder. There are three sub-layers of a layer in decoder. The bottom one is masked multi-head self attention, the upper two are multi-head attention and FFN which are similar to encoder. Residual connection followed by layer normalization will be performed after each sub-layer. Multi-head attention in the middle acts attention as in `seq2seq` model. Keys and values come from the output of encoder and query is from previous attention. This pattern allows the output of decoder based on the knowledge from encoder. Finally, we make some adjustment about masked skills in the self attention of each first sub-layers to prevent model knows the answer. Here, we need to make sure that the the i-th token can only attend to the token before i. Otherwise, the model can easily get the answer of $i+1$-th word by the input. Hence, we use a look-ahead mask to hide the information. The mask $L \in \mathbb{R}^{M\times M}$ is an lower triangular matrix with those elements equal to 1 included diagonal elements. Then L will combine with the padding mask $P \in \mathbb{R}^{M\times M}$. Masked multi-head self attention is just a multi-head self attention sub-layer with combination mask. The stack of $\mathrm{N}_d = 6$ layers of decoder in the original paper.

The output of final layer in the decoder will be input to a linear layer which switch it to the shape in $\mathbb{R}^{d_{\text{dict}}\times M}$ where d_{dict} is the total sub-words in the dictionary of our embedding way. After operating the softmax, we choose the corresponding index of the maximum values in the distribution. We sum up the total cross entropy of each position between the real answer and output, then update the model with Adam optimizer [16]. Transformer has three important contribution: First, it use neither CNN nor RNN to finish a NLP task. Second, self attention structure performs well. Finally, the model can be trained in a short time by the parallelizable computation.

5.3 Bidirectional Encoder Representations from Transformers

Bidirectional Encoder Representations from Transformers (BERT) is a new embedding model based on encoder of transformer [7]. Instead of the original embedding way, the contextual word representation creates the unique embedding for the tokens which depend on the context. BERT is also the fine-tuning model which all parameters in pretraind model can be change depend on the NLP tasks. These features make BERT reach the state-of-the-art in lots of tasks. In the following paragraph, we show more detain about BERT.

DOI:10.6814/NCCU202000709

There are two common methods of unsupervised NLP model: feature-based and fine-tuning. They both has a pre-train model which trained on a huge amount of unlabeled language data. It gets lots of knowledge of language. For feature-based model, we will fix all the parameters and add an output layer depend on our task. The task-specific model will be final model to solve the task. On the other hand, all the parameters will be adjusted to match the best result of our task in fine-tuning model. We have to modify input and output as well. Hence, BERT can be applied to many kinds of tasks. If we can train a powerful pre-trained model, we will get better performance easily. BERT teaches the model about language by two tasks: masked language model (MLM) and next sentence prediction (NSP). The input of pre-trained model will be a pair of sentences (A, B). We will randomly mask a token in each sequence. Model has to figures out the best choices of these two positions. Since the tokens have been hided, the attention are able to come from both directions (right-to-left and left-to-right) which lead to better result. At the meantime, the model has to judge whether sentence B is the next sentence of A. It can help the model get information about the relation between two sentences.

The input embedding of BERT is designed to fit these two tasks. First, we will divide our sentences to sub-words units as in Transformer. Second, we put some special token in the token pairs: <CLS> will be placed at each input pair, <SEP> locate in the end of sentence A and B, and <MASK> will be put to those hided tokens. Third, every token will have two kinds of embedding: the input embeddings are the corresponding index of each wordpieces, and segment embeddings which is the index for the sentence it belongs to. We set the maximum length of representation and finish padding here. Then each element in these two embeddings will be embedded to $\mathbb{R}^{d_{\text{model}}}$ dimension. Finally, the summation of these two embeddings will add position embeddings which are same as in Transformer and be the final input. The layer of BERT is the encoder in Transformer actually.

We call the i-th final output vector of BERT as the embedding of i-th input token. This representation is the contextual word representation which is different from the traditional embedding. The representation is based on context which is different from the one-to-one word embedding. We will also add a linearly classifier on each corresponding position of <CLS> and <MASK>. The linear classifier with sigmoid function will compute the relation of these two sentences. Others for <MASK> are just like the final output in Transformer: we get a distribution of all words in dictionary we use to split sub-words and pick the index of highest

DOI:10.6814/NCCU202000709

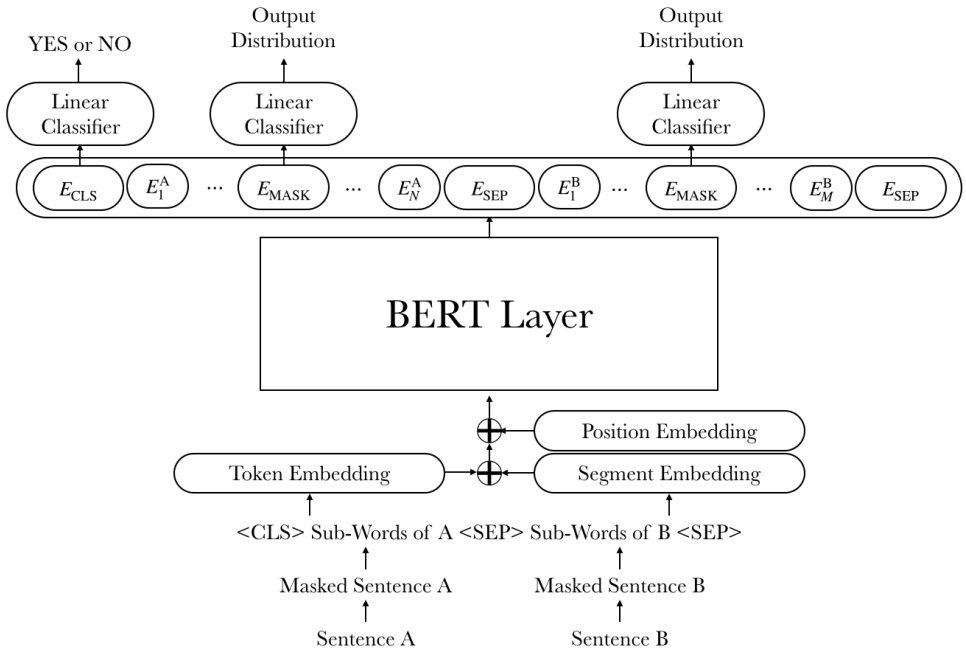

Figure 5.3: The architecture of pre-trained model in BERT

values as our output. We can compute loss and optimize our model. The pre-trained model is shown in Figure 5.3. Then we are able to use this pre-trained model by fine-tuning. We set up input and output type depend on our task and adjust all the parameters. Bert saves lots of cost to train a huge model. We can only use fine-tuning to obtain wonderful result. Bert also develops a strong method about word embedding.

DOI:10.6814/NCCU202000709

Chapter 6

Experiments

In this chapter, we will make prediction of numerical baseball data with LSTM and BERT. We also want to verify the difference of result between our models and baseball prediction systems which is called projection system as well. Here we introduce some famous projection system in MLB and their method.

6.1 Baseball Projection System

Projection system predicts all items of data about hitters and pitchers. There are two basic types of projection system in Major Leagur Baseball (MLB). Using previous data of the player to predict his future performance is one type. This method is very reasonable because we are used to judge a player by the performance in our mind. The Marcel the Monkey Forecasting System (Marcel) [36] is a basic system developed by Tom Tango. For each item, it sums up past three years performance with heavier weights for recent season and adjust the number about player's age and league average. The other type finds similar players to build the prediction. Player Empirical Comparison and Optimization Test Algorithm (PECOTA) [33] developed by Nate Silver uses historical data similar to the player and predict his performance by that trend. However, the definition about finding comparable players remains a secret. By the device updating, we have more and more detailed of baseball information. Steamer [6], which build by Jared Cross, Dash Davidson and Peter Rosenbloom, uses different way corresponding to each item to create projection by historical data. Moreover, it uses pitch-tracking data to help pitcher prediction. sZymborski Projection System (ZiPS) created by Dan Szymborski weights heavily

DOI:10.6814/NCCU202000709

for recent years which included velocity and pitch data for multiple years. Both are similar to Marcel. Furthermore, ZiPS also use comparable players to adjust the prediction which like PECOTA. THE BAT, developed by Derek Carty, some special information such as umpire and ball park factors are included in the stats while predicting [3]. In recent years, some popular projection system is based on the others projection system. One of the largest statistical analysis websites of baseball, FanGraphs, has published a new projection system called Depth Charts [8]. It combines the projection from ZiPS and Steamer and modify the result by the likely playing time. Average Total Cost Projection System (ATC) [5] which developed by Ariel Cohen uses a blend of projection system to predict the result. Each item takes unique combination of other projection. In this paper, we set ZiPS as the main comparison because its prediction data is easily accessible and precise.

6.2 Baseball Dataset Preparation

We collect players recorded in MLB with 20 features for every year during the period from 1998 to 2019. Since the final two teams Arizona Diamondbacks and the American Tampa Bay Devil Rays added in 1998, we started to collect data from that year. All players have at least 6 continuous years' experience in MLB which means they have at least 1 Plate Appearances (PA) for 6 continued seasons. These features include 17 annual accumulated performance and 3 player's status which are shown in Table 6.1.

Since home runs (HR) is the most effective way to score in the ball game, we set it as the main goal to predict in this paper. However, it is not only practically impossible but inefficient to predict the precision number of HRs. Therefore, we spilt HRs into disjoint subsets with every 5 HRs and each of them will be a class which is shown in Table 6.2. We choose these classes to be our outputs. For our data, we have 12 classes.

On the other side, we think 5-years-long data is stable enough to be input for model to know the player well. In conclusion, we use 5 continued years stat with 20 features for each year of a player to predict his HRs class in sixth year. Therefore, each data point will be

$$(\{\mathbf{x}_t\}_{t=1}^{5},\ y)$$

where $\mathbf{x}_t \in \mathbb{N}^{20}\ \forall\ t \in \{1, \cdots,\ 5\}$ denotes the t-th year stat of the player and $y \in C_i$ for some $1 \leq i \leq 12$ denotes the HR class of $\mathbf{x}_6$. We use 892 data points from 1998 to 2017 as training

DOI:10.6814/NCCU202000709

Code	Description
Age	Player's age
CM	Player's Height
KG	Player's Weight
G	Games Played
PA	Plate Appearances
R	Runs Scored
H	Hits
2B	Double Hits
3B	Triple Hits
HR	Home Runs
RBI	Runs Battle In
SB	Stolen Bases
CS	Caught Stealing
BB	Bases on Balls
SO	StrikeOuts
GDP	Grouned into Double Plays
HBP	Hit By a Pitch
SH	Sacrifice Hits
SF	Sacrifice Flies
IBB	Intended Bases on Balls

Table 6.1: Features used in this paper

data and 265 data points in 2018 and 268 datas in 2019 for testing which verify our model performance. The input of testing data points in 2018 is in the period of year 2013 to 2017 and output HR class is belonging to year 2018. Testing data points in 2019 are similar. ZiPS has 209 predictions in 2018 and 215 in 2019.

DOI:10.6814/NCCU202000709

Class	Numbers	Class	Numbers
C_1	0-4	C_7	30-34
C_2	5-9	C_8	35-39
C_3	10-14	C_9	40-44
C_4	15-19	C_{10}	45-49
C_5	20-24	C_{11}	50-54
C_6	25-29	C_{12}	55-59

Table 6.2: HRs classes

	LSTM	D/BN	TD	FC	D/BN	FC
A	128∗2	0.25	10	1024	0.25	128
B	128∗2	0.25		1024	0.25	128
C	128	0.25	10	1024	0.25	128
D	128∗2	0.25		1024	0.25	
E	128∗2	√	10	1024	√	128
F	32			1024		
G	64∗2	0.25		1024	0.25	128
H	128∗2	0.25		512	0.25	64
I	128					
J	128			1024	√	

Table 6.3: Architectures of LSTM models

6.3 Prediction Models

In this paper aim to use BERT to do the classification task by numerical data with time series. We compare its result to the prediction of LSTM and ZiPS. There are two kinds of pre-trained models of BERT we use: BERT-base and BERT-base-multilingual. Each of them has the version of cased and uncased which the wordpiece has the capital sub-words or not. We will fine-tune these pre-trained models by our task for 10 epochs. BERT-base has 12 layers, 768 hidden neurons and 12 attention heads for total 110 millions of parameters. It trained on English texts. BERT-base-multilingual-cased (BERT-mc) was trained on 104 languages based on Bert-cased and BERT-base-multilingual-uncased (BERT-mu) was trained on 102 languages

DOI:10.6814/NCCU202000709

based on Bert-cased. We set batch size for 14 with learning rate for 10^{-5}.

	MPA(%)	H1C(%)	L1C(%)	TPA(%)
LSTM A	43.8	14.3	14.0	72.1
LSTM B	45.7	12.8	12.5	71.0
LSTM C	43.0	14.0	15.0	72.0
LSTM D	45.3	11.3	13.2	69.8
LSTM E	40.0	9.8	6.8	56.6
LSTM F	48.7	13.6	12.5	74.8
LSTM G	44.9	13.2	13.6	71.7
LSTM H	41.5	14.7	15.1	71.3
LSTM I	45.3	12.8	13.2	71.3
LSTM J	42.6	9.4	9.4	61.4
ZiPS	25.4	11.0	37.3	73.7
BERT-c	46.8	12.8	13.6	73.2
BERT-u	49.1	9.4	13.6	72.1
BERT-mc	**50.6**	16.2	12.5	**79.3**
BERT-mu	47.6	16.6	12.1	76.3

Table 6.4: Prediction accuracy of 2018

On the other side, we build 10 kinds of model based on LSTM which shown in Table 6.3. Each model contains the LSTM layer and fully connected neural networks with BN or dropout(D). In LSTM column, $n * k$ means k layers with n neurons. In D/BN columns, number represents dropout rate and $\surd$ means BN is used instead of dropout. In model A, C and E, there is a timestep-wise dimension reduction (TD) in the middle which reduced the dimension. Finally, a standard fully connected layer with softmax will be our output. In all fully connected layers we use ReLU as the activation function to boost the training speed. The loss is counted by cross-entropy and optimizer is Adam with learning rate 10^{-8}. We set the batch size for 14 and trained for 20000 epochs. We use $(\{\mathbf{x}_t\}_{t=1}^5,\ y)$ be the input and output pair for LSTM. For BERT, we will combine all the data of $\{\mathbf{x}_t\}_{t=1}^5$ with space to be the string as the input. For example, $\{\mathbf{x}_t\}_{t=1}^5$ will be "22 23 14 $\cdots$ 0 162 89" and output y is the class.

DOI:10.6814/NCCU202000709

	MPA(%)	H1C(%)	L1C(%)	TPA(%)
LSTM A	**41.8**	11.9	13.4	67.1
LSTM B	38.4	12.3	14.2	64.9
LSTM C	36.9	14.6	15.7	67.2
LSTM D	38.1	14.2	14.6	66.9
LSTM E	36.9	10.5	9.3	56.7
LSTM F	40.3	13.8	13.8	**67.9**
LSTM G	37.7	13.8	14.9	66.4
LSTM H	34.7	16.4	15.3	66.4
LSTM I	38.8	11.6	13.4	63.8
LSTM J	39.6	6.3	10.0	55.9
ZiPS	30.1	16.0	19.7	65.8
BERT-c	39.2	12.3	14.9	66.4
BERT-u	37.7	9.3	17.9	64.9
BERT-mc	39.9	10.8	17.1	67.8
BERT-mu	40.3	13.4	14.2	**67.9**

Table 6.5: Prediction accuracy of 2019

6.4 Model Performance

Table 6.4 shows the maximum prediction accuracy (MPA) of data points in 2018. Higher for 1 class (H1C) means the prediction is the class C_{i+1} instead of the ground truth C_i. Lower for 1 class (L1C) is the opposite situation. These two values are valuable for the model performance. Finally, we sum up MPA, H1C and L1C to calculate the total prediction accuracy (TPA) of the model. The computation can be write as follow:

$$\text{MPA} = \frac{\text{Right Prediction}}{\text{Total Data Points}}$$
$$\text{H1C} = \frac{\text{Higher Prediction For 1 Class}}{\text{Total Data Points}}$$
$$\text{L1C} = \frac{\text{Lower Prediction For 1 Class}}{\text{Total Data Points}}$$
$$\text{TPA} = \text{MPA} + \text{H1C} + \text{L1C}$$

DOI:10.6814/NCCU202000709

The model performance of 2019 is presented in Table 6.5.

	MPA(%)	H1C(%)	L1C(%)	TPA(%)
LSTM A	42.2	7.5	17.2	66.9
LSTM B	41.4	9.0	13.8	64.2
LSTM C	39.6	11.9	14.2	65.7
LSTM D	42.8	7.8	13.8	63.4
LSTM E	36.9	8.2	13.4	58.5
LSTM F	40.3	10.1	15.7	66.1
LSTM G	39.6	7.1	16.0	62.7
LSTM H	40.3	8.6	14.6	63.5
LSTM I	40.0	12.3	14.6	66.9
LSTM J	36.7	10.8	10.8	58.3
ZiPS	30.1	16.0	19.7	65.8
BERT-c	39.6	12.3	15.7	67.6
BERT-u	39.2	13.4	17.9	**70.5**
BERT-mc	**44.4**	9.7	15.3	69.4
BERT-mu	42.5	10.1	15.3	67.9

Table 6.6: Prediction accuracy of 2019 after retrain

We can find that BERT models have marvelous performance on MPA in 2018. BERT-base-multilingual-cased can even reach 50 percent accuracy. Around 80% of player can be predicted in the list by the model. They show the ability that BERT can read about numerical data and performs well. LSTM models have more than 40% MPA, but TPA lies in a wide range. All models are able to predict more precisely than the predictions by ZiPS. However, ZiPS underestimates 37.3% of players by 1 class, its TPA still remain high. Only the model F of LSTM and BERT-base-multilingual models perform better than ZiPS in TPA. In conclusion, BERT-base-multilingual-cased is the best model in 2018. Although they performs well in 2018, the performance has dropped 5% to 10% while predicting the data points in 2019. Model A of LSTM gets highest MPA among all other models while both model F and BERT-base-multilingual-uncased have the greatest TPA. All models still outperform ZiPS in MPA with more models exceed the TPA of ZiPS. We think the reason about decreasing accuracy is that

DOI:10.6814/NCCU202000709

models may need more information about year 2018. Therefore, we add data points in 2018 into the original training data as the new one to retrain models and predict 2019 data points. The result is shown in Table 6.6. Almost every model has some growth in accuracy rate. BERT-base-multilingual-cased makes the leading position in MPA and BERT-base-uncased has the maximum percentage in TPA. BERT-base-multilingual always performs better than BERT-base. Using cased or uncased wordpiece embedding has no clear influence in the task. On the other hand, there is little difference between the simple struture of LSTM such as model F and complicated model like model A. Dropout is also better than BN here. Overall, BERT is a feasible and stable solution to predict numerical data. On the baseball prediction task, BERT is a new excellent projection system. It has the ability to know numbers and performs better than LSTM with less training time. Next, we analyze the prediction further.

	0-4	5-9	10-14	15-19	20-24	25-29	30+
LSTM A	79	14	19	0	3	1	0
LSTM B	79	18	15	0	9	0	0
LSTM C	74	17	19	0	3	1	0
LSTM D	79	21	15	0	4	1	0
LSTM E	83	6	4	**4**	4	**4**	**1**
LSTM F	79	22	16	0	11	1	0
LSTM G	77	21	17	0	3	1	0
LSTM H	71	20	18	0	0	1	0
LSTM I	80	17	17	0	6	0	0
LSTM J	88	2	7	3	11	2	0
BERT-c	83	**22**	8	0	11	0	0
BERT-u	**89**	17	14	1	8	1	0
BERT-mc	84	17	**23**	0	9	1	0
BERT-mu	81	17	11	0	**17**	0	0
Ground Truth	107	43	41	25	27	8	14

Table 6.7: Number of correct predictions in 2018

DOI:10.6814/NCCU202000709

	0-4	5-9	10-14	15-19	20-24	25-29	30+
LSTM A	72	14	**18**	0	7	1	0
LSTM B	72	13	7	0	11	0	0
LSTM C	68	14	15	0	1	1	0
LSTM D	72	15	11	0	3	1	0
LSTM E	77	7	7	0	4	**2**	**2**
LSTM F	72	13	13	0	9	1	0
LSTM G	70	**18**	10	0	1	**2**	0
LSTM H	64	17	10	0	0	**2**	0
LSTM I	72	11	14	0	7	0	0
LSTM J	**81**	1	12	**1**	10	0	1
BERT-c	74	17	4	0	10	0	0
BERT-u	77	10	8	**1**	4	1	0
BERT-mc	76	12	12	0	6	1	0
BERT-mu	68	15	10	0	**15**	0	0
Ground Truth	102	35	39	24	28	13	27

Table 6.8: Number of correct predictions in 2019

6.5 Class Result

In this part, we show more detail of the prediction. In Table 6.7, we can see the distribution of prediction for each class in 2018. The head of columns represents the number of HRs in each class. Obviously, we are able to observe that the first three class, which under 15 HRs, and C_5 class are easily predictable. However, C_4 and those classes more than 25 HRs are hard to predict. The result of 2019 and retrain model which shown in Table 6.8 and Table 6.9 represent similar pattern. We think that for C_4 class, the data of player is similar to C_3 and C_5 which lead model to ignore this class. For the class which more than 25 HRs, the data points for training is less and varies a lot. Hence, we will try other ways to predict these classes to enhance the accuracy. ZiPS result in Table 6.10 shows that its prediction is cross every class but the accuracy of each of them are low.

DOI:10.6814/NCCU202000709

	0-4	5-9	10-14	15-19	20-24	25-29	30+
LSTM A	**81**	18	4	0	9	**1**	0
LSTM B	77	16	10	0	8	0	0
LSTM C	74	11	12	0	9	0	0
LSTM D	80	14	11	0	7	0	0
LSTM E	79	11	7	1	1	0	0
LSTM F	76	13	12	0	7	0	0
LSTM G	78	11	9	0	8	0	0
LSTM H	79	13	7	0	9	0	0
LSTM I	75	8	**14**	0	9	**1**	0
LSTM J	**81**	2	4	**5**	3	**1**	**2**
BERT-c	74	19	8	1	3	**1**	0
BERT-u	71	**22**	4	0	7	**1**	0
BERT-mc	75	18	12	0	**13**	**1**	0
BERT-mu	73	22	12	0	6	**1**	0
Ground Truth	102	35	39	24	28	13	27

Table 6.9: Number of correct predictions in 2019 after retrain

	0-4	5-9	10-14	15-19	20-24	25-29	30+
ZiPS 2018	14	7	12	7	6	2	5
Ground Truth 2018	107	43	41	25	27	8	14
ZiPS 2019	15	17	14	6	7	2	3
Ground Truth 2019	102	35	39	24	28	13	27

Table 6.10: Number of correct predictions of ZiPS

DOI:10.6814/NCCU202000709

Chapter 7

Conclusion

In this paper, we research the possibility of using numerical data with order to be the input of the powerful NLP model BERT. We predict the class of HRs number of MLB players by their past performance. Since there is no clear track of this data to be forecast, classification task will be a big challenge for BERT. We also compare the result for the traditional recurrent deep learning model LSTM and the well-known baseball projection system ZiPS. Performance of BERT and LSTM are amazing. All models have outperformed ZiPS in the maximum prediction accuracy. Moreover, models based on BERT are usually have better performance than those based on LSTM. Although it performs well, there are two phenomenon we should notice. BERT is still easily focus on the majority of the training class. Meanwhile, data have to keep updating or BERT will have less information to predict. In conclusion, BERT is a feasible and effective way to predict output by numerical input. Self attention skill helps model make more knowledge about data sequence. We can actually use this NLP model in more field such as finance and natural science. There are several potential research direction in this classification problem:

1. The structure of BERT can be adjusted due to numerical data input to increase accuracy.
2. Other new stats can be added such as batting average on balls in play (BABIP) or average ball exit velocity (EV).
3. Creating an embedding for baseball player to know more information about them.
4. Solving the prediction result of imbalanced data.

DOI:10.6814/NCCU202000709

Bibliography

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2014.

[3] Derek Carty. The bat. www.RotoGrinders.com.

[4] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

[5] Ariel Cohen. Atc. www.fangraphs.com.

[6] Jared Cross, Dash Davidson, and Peter Rosenbloom. Steamer projections. steamerprojections.com/.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[8] FanGraphs. Depth charts. www.fangraphs.com.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[10] Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*. Studies in Computational Intelligence. Springer, Berlin, 2012.

DOI:10.6814/NCCU202000709

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2016.

[12] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors, 2012.

[13] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[14] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[15] Anil K. Jain, Jianchang Mao, and K. Mohiuddin. Artificial neural networks: A tutorial. *IEEE Computer*, 29:31–44, 1996.

[16] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[17] Kaan Koseler and Matthew Stephan. Machine learning applications in baseball: A systematic literature review. *Applied Artificial Intelligence*, 31(9-10):745–763, 2017.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[19] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, volume 86, pages 2278–2324, 1998.

[20] J. Y. Lettvin, H. R. Maturana, W. S. McCulloch, and W. H. Pitts. What the frog's eye tells the frog's brain. *Proceedings of the IRE*, 47(11):1940–1951, 1959.

[21] Arlo Lyle. *Baseball prediction using ensemble learning*. PhD thesis, University of Georgia, 2007.

DOI:10.6814/NCCU202000709

[22] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[23] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[24] M. Minsky and S. Papert. *Perceptrons*. MIT Press, Cambridge, MA, 1969.

[25] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In Johannes Fürnkranz and Thorsten Joachims, editors, *ICML*, pages 807–814. Omnipress, 2010.

[26] Andrew Y. Ng. Feature selection, l1 vs. l2 regularization, and rotational invariance. In *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML '04, page 78, New York, NY, USA, 2004. Association for Computing Machinery.

[27] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[28] Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, 2018.

[29] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2016.

[30] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2016.

[31] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, 1986.

[32] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam,

DOI:10.6814/NCCU202000709

Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[33] Nate Silver. Introducing pecota. *Baseball Prospectus*, 2003:507–514, 2003.

[34] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[35] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[36] Tom Tango. Marcel. www.tangotiger.net.

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[38] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989.

[39] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation, 2016.

[40] Xue Ying. An overview of overfitting and its solutions. *Journal of Physics: Conference Series*, 1168:022022, feb 2019.

DOI:10.6814/NCCU202000709