Final Project Report

Kaggle Competition: Time Series Classification Using Large Synoptic Survey Telescope (LSST) Dataset

Kaggle Username: pngo1997

Public score: 1.47044

Word count: 2355

Mai Ngo

DSC 578 Neural Network and Deep Learning - SEC 701

Dr. Noriko Tomuro

November 21, 2023

Table of Contents

Executive Summary

This report entails the journey of my completion for the final project of course CSC 578: Neural Networks and Deep Learning. The project is a Time Series Classification using Recurrent Neural Network (RNN) with Large Synoptic Survey Telescope (LSST) dataset. There will be five main parts of this report.

- ✍ Data preprocessing which will be explained thoroughly my logic and approaches to two versions of the training dataset that will be feed to the RNN model – original and down sampled (first further advanced/development step).
- ✍ A breakdown of the first model which I deemed as a breakthrough indicates that my codes are working correctly, and I can start enhancing my project.
- ✍ The second model which ultimately is determined as the best model, yields the highest score in Kaggle competition of 1.47044.
- ✍ Second further/advanced development step of using Keras Tuner – an automatic scalable hyperparameter optimization framework that has tremendously help me with this project.
- ✍ Finally, my reflection towards this project journey. Also, what I have gained throughout this course.

Ultimately, by doing this project from scratch without starter code provide. I was able to utilize my logic and code flow independently. Significantly enhance my ability in data manipulation and deep learning algorithm knowledge. I also want to point out that the hints you provided us are extremely helpful. Without them, I would easily get lost in completing this project.

Data Preprocessing

Original Data

To set up my base code works correctly – which means that I can generate a prediction csv file and successfully submit to Kaggle, I followed all the instructions/hints you provided meticulously. First, I imported both training and testing data as pandas data frame. Then decided to separate the 'target' column individually by itself – this is my usual approach whenever I do independent work. I also do data exploration, noticeably visualizing the 'target' class frequency distribution. The plot shows that there is data imbalance issue, i.e., class 'c-90' is has ten times more observation count than class 'c-95'; 1088 and 72 counts, respectively (Figure 3). At this point, I decided to keep the original training data as it is and focusing on making my code functional first.

Moving on to process 'target' column which convert to integers. This was pretty straightforward as it is. Same as code for reshaping the predictors. After reshaping, in training data, total number of rows stay the same: 3356. Given each row now is 2D array with dimensions (timeSteps = 6, features = 36). The only thing I added extra is checking that my reshaped data are holding values correctly compared to the original data. After this step, I decided to split the training data into train and validation using 0.8/0.2 split ratio. Furthermore, the split was executed using row index and in top-down order just to make sure if there is any time series nature between rows, it can be remained. After splitting, the shapes are:

```
After split:
X Train shape: (2684, 6, 36)
Target Train shape: (672, 6, 36)
X Validation shape: (2684, 1)
Target Validation shape: (672, 1)
```

Figure 1. Shapes of training and validation sets after split – original data.

In addition, I am also checking that after the split, making sure all of 11 'target' classes are existed in both training and validation sets. My expectation both sets should also have the same ratio class distribution. Indeed, the expectation holds, and I decided to move forward training the model with train and validation data.

```
print("Class Distribution in Training Set:")
class_distributionTrain_table
```

Class Distribution in Training Set:

Class	8	2	5	1	4	0	7	9	6	3	10
% in Training Set	32.303	16.244	12.891	10.991	6.148	5.365	5.328	3.204	2.906	2.571	2.049

```
print("\nClass Distribution in Validation Set:")
class_distributionVal_table
```

```
Class Distribution in Validation Set:
```

Class	8	2	5	1	4	0	7	9	3	6	10
% in Validation Set	32.887	14.583	13.69	12.351	7.292	4.315	3.869	3.274	2.679	2.53	2.53

Figure 2. Class ratio distribution (%) of training and validation sets.

Further/Advanced Development - Down Sampling Data

After ensuring my starter code successfully work. I decided to make two further advanced developments. The first one was to tackle class imbalance issue. There are many ways to handle this and since I want to explore new method, I decided to use Google Developers - Machine Learning Imbalanced Data reference (Google Developers, 2023). According to the article, there are two steps:

Step 1: Down sample the majority classes.

From the original 'target' class distribution plot (Figure 3.), class 'c-90' has the most: 1088 counts, while class 'c-95' has the least of 72 counts. That is roughly 1 of 'c-95' to 15 of 'c-90'. Same logic applies to other classes in between. Given this, for 'c-90', apply down sampling by a factor of 10 to improve the balance which now, 1 of 'c-95' to approximately 1.5 of 'c-90'. Apply the same down sampling factor logic to other classes as well. Expecting afterward, each class has around 100 observations.

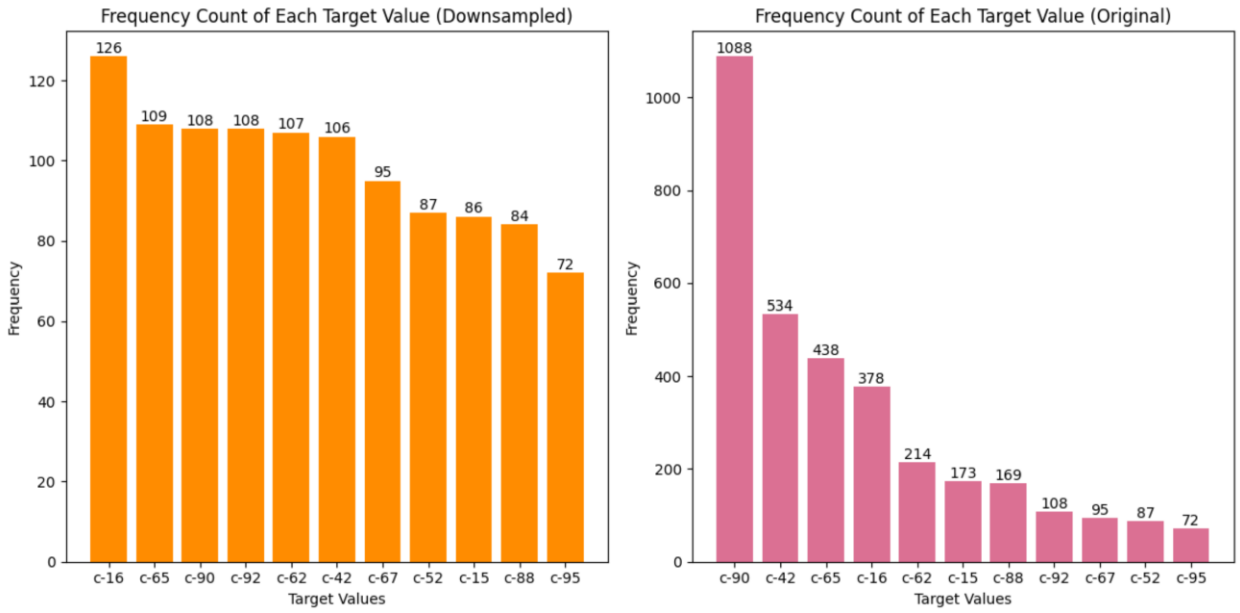

Figure 3. Frequency count of each 'target' class before and after down sampling on full training data.

```
downsamplingFactors = {'c-90': 10, 'c-42': 5, 'c-65': 4, 'c-16': 3, 'c-62':
2, 'c-15': 2, 'c-88': 2, 'c-92': 1, 'c-67': 1, 'c-52': 1, 'c-95': 1}
```

I want to explicitly clarify that I applied down sample on the original training data with 3,356 observations. After this step, down sampled data yields 1,088 observations. Figure 3. above shown the frequency distribution of 'target' class before and after. The ratio is exactly like what I expected, roughly 100 observations per 'target' class. After this step, I applied the same code flow as I did with the original data before. After splitting into training and validation sets, the shapes are:

```
After split:
X Train shape: (870, 6, 36)
Target Train shape: (218, 6, 36)
X Validation shape: (870, 1)
Target Validation shape: (218, 1)
```

Figure 4. Shapes of training and validation sets after split – down sampled data.

Step 2: Upweight the down sampled classes.

Since we down sampled class 'c-90' by a factor of 10, we will apply that factor as respective weight contribution to each instance of class 'c-90' during training. So, each retained observations will now get higher importance weight (originally 1). This compensates for the number of instances got taken out, ensuring that the model pays more attention to the instances that are still present. We will apply this during model fit using the above **downsamplingFactors** dictionary as reference through later **model.fit** which has **'class_weight'** parameter allows to specify 'target' class contribution.

As for testing data, the process was very straightforward as it is. I only need to reshape it using the same code for training then make prediction. The csv file was written given first column 'id' represent row index of each observation in test set.

First Model

First model was used to make sure my codes are functional using the original training data. I initially started out with the model you provided in the hint, one Long short-term memory (LSTM) layer of 8 units. I also decided to use Sequential API instead of Functional API because I am more familiar with it, we have been using this framework in both homework 4 and 7. Establishing the model code was not that difficult for me as you have provided us hints where to apply normalizer, input shape, and 'return_sequence = True' if I want to stack recurrent layers.

The initial performance result was not that great since it was a very simple model to start with. I also notice the lowest performances in worst case scenario are around 32% accuracy on both training and validation sets. Then I tested with four different recurrent layers (RL), all sort of combinations: LSTM, SimpleRNN, Bidirectional and Gated Recurrent Unit (GRU). Ultimately, I notice that model using Bidirectional(LSTM) recurrent layer as first layer yields the best promising results. I also added two dense layers of size 256, dropout percent of 0.3; and 128 size, dropout percent of 0.25, respectively. Both using ReLU activation functions. I find this combination of dense layer gives me consistent result to play with the recurrent layers, so I decided to keep it as it is. Using 512 or 64 size would cause the performance output to fluctuate way too much. Also, learning rate stays the same as 0.001; I notice the higher the learning rate, accuracy score drops significantly. Regularizations are also not ideal in my opinion, I tried with all three kernel/bias/activity, both L1 and L2 regularizations with different combinations and also see the model performance decreases.

Eventually, first model gives me 50% accuracies and 1.4 on losses on both train and validation sets. First model architecture:

4 Recurrent Layers:

- ✍ Bidirectional LSTM layer with 32 units.
- ✍ Bidirectional GRU layer with 96 units.
- ✍ Simple RNN layer with 96 units.
- ✍ Bidirectional LSTM layer with 32 units.

2 Dense Layers:

- ✍ Dense layer of size 256 with ReLU activation function and dropout percent of 0.3.
- ✍ Dense layer of size 128 with ReLU activation function and dropout percent of 0.25
- ✍ Output layer with Softmax activation for classification, size 11.

Second Model – Best Model

The second model which is my best model was built based on the first model. Additionally, I used automatic hyperparameters turning Keras Tuner – will be discussed later to find the optimal combination of hyperparameters to use. At this point, from many trials and errors, I already have some pre-requisite for several parameters: learning rate of 0.001 using Adam optimizer, no regularization applies, fixed Dense layers, and dropout implementation after each RL. This model gives me the highest-ranking score on Kaggle leader board using the original training data. Second model architecture:

5 Recurrent Layers:

- ✍ Bidirectional LSTM layer with 512 units.
- ✍ Simple RNN layer with 64 units.
- ✍ Bidirectional GRU layer with 32 units.
- ✍ Simple RNN layer with 64 units.
- ✍ LSTM layer with 512 units.

2 Dense Layers:

- ✍ Dense layer of size 256 with ReLU activation function and dropout percent of 0.3.
- ✍ Dense layer of size 128 with ReLU activation function and dropout percent of 0.25
- ✍ Output layer with Softmax activation for classification, size 11.

The optimal model yields an accuracy of 51.19% on the validation set and 52.72% on the training set, with comparable losses of 1.3626 on the validation and 1.3736 on the training set.

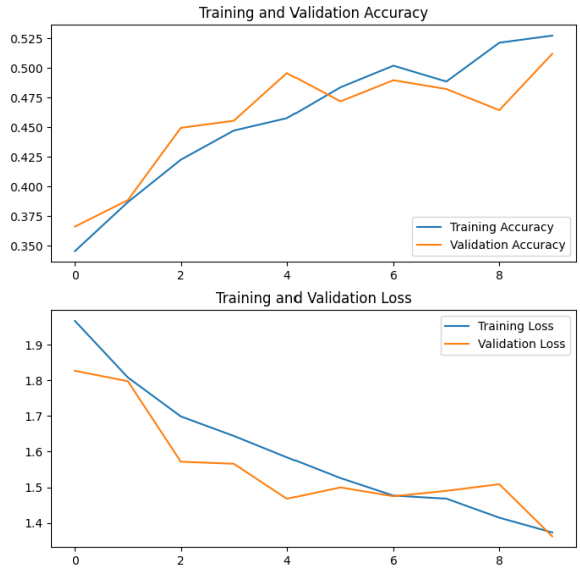

Figure 4. Accuracy and Loss on training and validation sets – best model.

Automatic Hyperparameters Keras Tuning

This was the most interesting part of my final project; I really enjoyed this. Initially, I was manually doing grid search and after a while, it was not effective as running time got super slow. After researching some automatic hyperparameters options, I came across Keras Tuner – a very straightforward and convenient hyperparameter search framework (Keras, n.d.). Basically, I just need to input the full code of my neural network framework given as a function, start from initiate Sequential model up to the final output layer with Softmax activation function. Additionally, Keras Tuner requires to specify the minimum, maximum, and steps of how many units I want to tune within each respective layer.

This had given me so much room to play with the model. Using both the original and down sampled training data. I tried with Conv1D layer, but the performance result was not that good. Then I also implemented different activation functions, batch sizes, dropout rate, regularization, some other optimizers, all kind of combinations I can think of. In addition, by doing this project on Google Colab, I changed the accelerator of GPU and was able to run the tuner countless of time efficiently. Figure 5. below shows how Keras Tuner works, I need to input the number of trials, and it will retain the best combination of hyperparameters, eventually fit the model using that best combination.

```
Trial 4 Complete [00h 01m 34s]
val_accuracy: 0.17431192100048065

Best val_accuracy So Far: 0.22477063536643982
Total elapsed time: 00h 09m 04s

Search: Running Trial #5

Value             |Best Value So Far |Hyperparameter
480               |256               |lstm_filters
416               |160               |rnn_filters

Epoch 1/10
109/109 [==============================] - 40s 266ms/step - loss: 6.6512 - accuracy: 0.1379 - val_loss: 2.6488 - val_accuracy: 0.0688
Epoch 2/10
109/109 [==============================] - 25s 231ms/step - loss: 6.3419 - accuracy: 0.1471 - val_loss: 2.5757 - val_accuracy: 0.0963
```

Figure 5. Automatic Hyperparameters tuning using Keras Tuner.

Ultimately, my goal was to see if the down sampled data would perform well, but it was not. No matter how much combination I use, the furthest accuracy score I can yield is roughly 26%, very poor performance compared to original training data. The prediction score on Kaggle using this model was roughly 1.7, surprisingly not that far from my best model. As to implement **Step 2: Upweight the down sampled classes.** This is the dictionary used for **'class_weight'** parameter, given keys are class integers:

```
upWeight = {0: 2.0, 1: 3.0, 2: 5.0, 3: 1.0, 4: 2.0, 5: 4.0, 6: 1.0, 7: 2.0,
8: 10.0, 9: 1.0, 10: 1.0}
```

Reflection

Overall, this project definitely was a useful experience for me trying new things and using my own creative mind. I love project like this where I can find new things and have the freedom to implement it, like Keras Tuner. As for the data, I can conclude that the original data performs better than the down sampled data, perhaps I need to try different data balancing techniques. By doing this project step by step following your hints, I find it not as stressful. Also, when you confirmed that the final project framework is similar to Homework 7, it helps me a lot in visualizing what to do, and make sure each of the baby steps were done correctly.

For this course, I was able to learn and practice neural network thoroughly. I actually understand every concept and homework. I have made a lot of useful notes and neural network graph that get me through this class, and I am really proud that I was able to compile those important material. My biggest lesson from this class is for any project, it is better to understand theoretical part first, pen and paper, then start writing code. Furthermore, honestly, I wish that I did not take this class and Big Data Mining together, because then I will enjoy this class much more. After homework 5, I feel like everything was done like a chore and I did not really get excited in doing homework and learning new things. I am glad I took this class with you because again, I like how much you challenge us which I have mentioned to you countless of time and improving my coding skill and logic. Finally, I think my debugging skill have shot up after this class, thanks to Homework 5!

References

Google Developers - Machine Learning. (2023, June 9). Imbalanced Data. https://developers.google.com/machine-learning/data-prep/construct/sampling-splitting/imbalanced-data

Keras. (n.d.). KerasTuner. https://keras.io/keras_tuner/