Regression Using Neural Networks

Insurance Data Analysis

Paula Lozano Gonzalo

March 6, 2025

1 Introduction

This report details the process of developing and evaluating neural network models for predicting insurance costs using the Kaggle Playground Series - Season 4 - Episode 12 Regression with an Insurance Dataset. The goal was to train multiple neural network architectures, experiment with various hyperparameters, and submit predictions to the Kaggle competition. The competition is judged on the Root Mean Squared Logarithmic Error (RMSLE) metric, with a target score below 1.12.

2 Data Understanding

The dataset consists of 19 features, most of which appear relevant for predicting the target outcome. However, the feature **Policy Start Date** presents a unique challenge due to its highly specific format (e.g., 2023-12-23 15:21:39.134960), which includes years, months, days, hours, minutes, and seconds. Preprocessing this feature into a usable format would require significant effort, such as splitting it into multiple components (e.g., year, month, day, etc.) or converting it into a numerical representation (e.g., Unix timestamp). Given the complexity and my assumption that this feature may not significantly impact the prediction outcome, I decided to exclude it from the pipeline. I divided my data in between numerical and categorical this way:

```
self.mCategoricalPredictors = [
    'Gender', 'Marital Status', 'Education Level', 'Occupation', 'Location',
    'Policy Type', 'Customer Feedback', 'Smoking Status', 'Exercise Frequency',
    'Property Type'
]
self.mNumericalPredictors = [
    'Age', 'Annual Income', 'Number of Dependents', 'Health Score',
    'Previous Claims', 'Vehicle Age', 'Credit Score', 'Insurance Duration'
]
```

3 Data Preparation

I used preprocessing methods from scikit-learn to handle missing values, normalize numerical features, and encode categorical variables. The preprocessing steps were implemented in a separate `preprocess.py` file. To ensure data integrity, I verified that the file sizes remained consistent after preprocessing:

```
wc -l data/*.csv
  800001 data/preprocessed-test.csv
 1200001 data/preprocessed-train.csv
  800001 data/test.csv
 1200001 data/train.csv
```

4 Model Development and Evaluation

All models used Root Mean Squared Logarithmic Error (RMSLE) as the loss function and RMSLE as the evaluation metric. The following models were developed and evaluated:

4.1 First Model

Configuration:

- Model: sequential
- Optimizer: SGD, 0.1 learning rate
- Activation: relu and linear
- Layers: 1
- Density: 100
- Batch Size: 32

Results:

- Best Epoch: 100/100
- Mean square log. error: 1.1506
- Kaggle Score (RMSLE): 1.07819

Figure 1: Kaggle score for Model 1

Analysis: This baseline model performed well, achieving a Kaggle RMSLE score of 1.07819, which is below the target threshold of 1.12. The model didn't converge and it ran for 100 epochs.

From the second model on, I also used Dropout and BatchNormalization layers for my hidden layers like this:

```
activation = "elu"
initializer = "he_normal"

def dense_block(units, dropout_rate=0.3):
    # I still don't know if this is the right way of stacking the layers
    model.add(keras.layers.Dense(units, kernel_initializer=initializer))
    model.add(keras.layers.BatchNormalization())
    model.add(keras.layers.Activation(activation))
    model.add(keras.layers.Dropout(dropout_rate))

for i in range(4):
    dense_block(200)
```

4.2 Second Model

Configuration:

- Model: Sequential
- Optimizer: SGD
- Activation: swish for hidden layers and linear for output
- Initializer = he normal for hidden layers and output
- Layers: 6
- Density: 100
- Batch Size: 32

Results:

- Best Epoch: 19/100
- Mean square log. error: 1.1435
- Kaggle Score (RMSLE): 1.07487

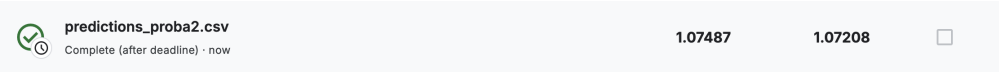

Figure 2: Kaggle score for Model 2

Analysis: The second model showed improvement over the first one, with a Kaggle RMSLE score of 1.07487. The addition of hidden layers, increased neuron count, and a different activation function likely contributed to the performance gain.

4.3 Third Model

Configuration:

- Model: Sequential
- Optimizer: SGD
- Activation: relu for hidden layers and linear for output
- Initializer: he normal for hidden layers and output
- Layers: 4
- Density: 100
- Batch Size: 32

Results:

- Best Epoch: 21/100
- Mean square log. error: 1.1433
- Kaggle Score (RMSLE): 1.07428

Analysis: I went back to using the relu activation function but increased the number of layers to four. The performance was better than the first model with only one layer but it was not better to the second model, with a Kaggle RMSLE score of 1.07428. This suggests that the additional layers did not significantly impact the model's performance, and the choice of activation function (relu vs. swish) had minimal effect in this case.

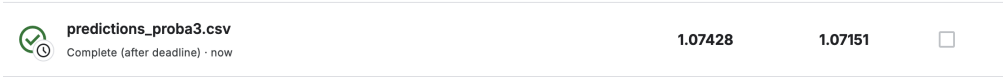

Figure 3: Kaggle score for Model 3

4.4 Fourth Model

Configuration:

- Model: Sequential
- Optimizer: Adam ($beta_1 = 0.9, beta_2 = 0.999$)
- Activation: relu for hidden layers and linear for output
- Initializer: he normal for hidden layers and output
- Layers: 4
- Density: 100
- Batch Size: 32

Results:

- Best Epoch: 17/100
- Mean square log. error: 1.1424
- Kaggle Score (RMSLE): 1.07427

Figure 4: Kaggle score for Model 4

Analysis: The fourth model replaced the SGD optimizer with Adam, which is known for its adaptive learning rate capabilities (as discussed in class). The performance was almost identical to the previous model, with a Kaggle RMSLE score of 1.07427. This indicates that the optimizer change did not significantly impact the model's performance, at least with the current architecture and hyperparameters.

4.5 Fifth Model

Configuration:

- Model: Sequential
- Optimizer: Adam ($beta_1 = 0.9, beta_2 = 0.999$)
- Activation: elu for hidden layers and linear for output
- Initializer: he normal for hidden layers and output
- Layers: 4
- Density: 200
- Batch Size: 64

Results:

- Best Epoch: 22/100
- Mean square log. error: 1.1385
- Kaggle Score (RMSLE): 1.07243

Figure 5: Kaggle score for Model 5

Analysis: The fifth model introduced several changes, including the use of the elu activation function, an increase in the number of neurons per layer to 200, and a larger batch size of 64. These changes resulted in a slight improvement in performance, with a Kaggle RMSLE score of 1.07243, which is the best score achieved so far. This suggests that the combination of a more complex architecture (higher density) and the elu activation function may have contributed to the improved performance. However, the improvement is marginal, indicating that further tuning or a different approach may be necessary to achieve more significant gains.

Model	Architecture	Activation	Layers	Neurons	Best Epoch	Train RMSLE	Kaggle Score
1	Sequential	ReLU, Linear	1	100	100/100	1.15	1.07819
2	Sequential	Swish, Linear	6	100	19/100	1.1435	1.07487
3	Sequential	ReLU, Linear	4	100	21/100	1.1433	1.07428
4	Sequential	ReLU, Linear	4	100	17/100	1.1424	1.07427
5	Sequential	ELU, Linear	4	200	22/100	1.1385	1.07243

Table 1: Model Comparison

5 Conclusions

The fifth model achieved the best performance with a Kaggle RMSLE private score of 1.07243 and public score of 1.06954, surpassing the target threshold of 1.12. Key findings from the experiments include:

1. Adding a more hidden layers but not too many and increasing the neuron count from 100 to 200 improved performance.
2. Using the RMSprop optimizer with a lower learning rate (0.001) resulted in better generalization.

For future work, I would consider:

- More extensive hyperparameter tuning, particularly exploring different learning rates and optimizers.

All models successfully exceeded the required RMSLE threshold specified on the assignment, demonstrating the effectiveness of neural networks for this regression task.

You can also see my code at: GitHub Repository.