CSE 151B Project Final Report

Jiancheng Liang
Department of Mathematics
University of California, San Diego
San Diego, CA 92122
jil269@ucsd.edu

1 Task Description and Background

The deep learning task is to develop a model that can predict the travel time of taxi rides based on partial trajectory information[1]. This involves training a deep learning algorithm to analyze the available data and make accurate predictions about how long a given taxi trip will take to complete. The task is important because it can significantly improve transportation efficiency and customer satisfaction[18]. By accurately predicting the travel time, a central dispatch system can efficiently assign taxis to pickup requests, which means reduced waiting times for passengers and improved overall service quality. It also helps taxi drivers by enabling them to plan their schedules better and maximize their earnings[2].

Here are a few real-world examples where accurate travel time prediction can make a significant impact: (1) Efficient Ride-Hailing Services: Some ride-hailing platforms such as Uber and Lyft heavily rely on predicting travel times to match passengers with available drivers. Accurate predictions enable these platforms to minimize waiting times, increase customer satisfaction, and optimize driver utilization. (2) Traffic Management: Traffic congestion is a major problem in many large cities worldwide. By accurately predicting travel times, traffic management systems can proactively identify congested areas and suggest alternate routes to drivers, reducing traffic congestion and improving overall traffic flow. (3) Public Transportation Planning: Accurate travel time predictions can help public transportation authorities optimize bus and train schedules, which allows them to adjust the frequency of services based on anticipated demand and minimize delays, leading to more efficient and reliable public transportation systems. (4) Emergency Services: In emergency situations, such as medical emergencies or natural disasters, accurately predicting travel times becomes critical for emergency response services because it allows them to plan routes and allocate resources effectively, potentially saving lives by minimizing response times.

1.1 Problem description

Our deep learning task is a regression task that given an input feature X, which denote to the partial trajectory information of a taxi ride (e.g., call type, poly-line, day type, taxi id), we apply deep learning model f to predict the output travel time of the taxi ride Y. Let's assume we have a dataset D consisting of N examples, where each example consists of an input-output pair (X_i, Y_i). The goal is to find a function f that maps the input X_i to the predicted output Y_i. We can formulate the prediction task as an optimization problem by minimizing the discrepancy between the predicted travel time Y_i and the actual travel time Y_i^* for each example in the dataset. The loss function is a suitable loss function that quantifies the difference between the predicted travel time $f(X_i)$ and the actual travel time Y_i^*, in this study, we choose MSE loss as our loss function. The optimization formula is:

[1]https://www.kaggle.com/competitions/ucsd-cse-151b-class-competition/

[2]Code available: https://github.com/K-Liang-6/travel_time_pred/

$$\min \frac{1}{N} \sum_{i=1}^{N} ||Y_i, f(X_i)||_2^2 \tag{1}$$

Real-world examples Regarding the potential of the model to solve other tasks beyond this project, deep learning models have shown versatility and generalizability in solving a wide range of tasks across various domains[11]. Although the specific architecture and training may need to be adapted, the underlying principles can be applied to similar prediction problems. Here are some examples where the model can potentially be extended: (1) Traffic Flow Prediction[12]: By incorporating additional data sources such as historical traffic patterns, road conditions, and weather information, the model can be adapted to predict traffic flow at different locations and times, aiding in traffic management and route planning. (2) Demand Prediction for Ride-Sharing Services: By considering factors like time of day, day of the week, and location, the model can be modified to predict the demand for ride-sharing services in different areas, helping platforms optimize their driver allocation and pricing strategies. (3) Travel Time Estimation for Autonomous Vehicles[13]: Autonomous vehicles require accurate travel time estimation to plan their routes and make real-time decisions. By adapting the model to incorporate sensor data and road conditions, it can be utilized for travel time estimation in autonomous driving scenarios.

2 Exploratory Data Analysis

There are 1710670 training data and 320 test data. The training data includes 9 columns, including 8 raw features, such as *trip id*, *call type*, *origin call*, *origin stand*, *taxi id*, *timestamp*, *day type*, *missing data* and one target feature *polyline*. Test data only includes 8 raw features without the target feature *polyline*.

2.1 Data description

Trip id is the unique identifier for each trip, *call type* is the category of the ride, if is ‘A’, this trip was dispatched from the central, if is ‘B’, the trip was demanded directly to a taxi driver on a specific stand, if is ‘C’, it indicates that the trip was demanded on a random street or otherwise. *Origin call* refers to the unique identifier for the phone number to call the taxi when the *call type* is ’A’. *Taxi id* is the identifier for the taxi to take this ride. *Day type* is the daytype of the trip’s start. When *day type* is ’A’, it identifies the trip started on a normal day or weekend, when *day type* is ’B’, it indicates that the trip started on a holiday or other special day, when *day type* is ’C’, it shows that the trip started on a day before a type-B day. *Timestamp* is the timestamp that identifies the trip’s start. *Polyline* is a list of GPS coordinates (i.e. WGS84 format) mapped as a string. The beginning and the end of the string are identified with brackets (i.e. [and]). Each pair of coordinates is also identified by the same brackets as [LONGITUDE, LATITUDE]. The coordinates were recorded every 15 seconds during the trip. The first item represents the starting point and the last item corresponds to the destination. We visualize the training data in Figure 1.

2.2 Data prepossessing

First, we apply random sampling method to randomly partition the data while ensuring a representative distribution of samples in both sets. Specifically, we used *sklearn.model_selection.train_test_split()* function[16] to split our training and validation set, 20 % of the data is randomly allocated to the validation set while the remaining portion is used for training.

Second, since the original features only contains 8 columns, which is not enough to predict a large regression model, we instead apply several feature engineering methods. For example, (1) Extracting Time-based Features: from the *timestamp* feature, we extract various time-related features such as hour of the day, day of the week, month, etc. We also construct the sin and cos values of these features to make the variables continuous, as well as calculating some higher-level categorical features, such as season, time period, etc. These features can capture any time-based patterns or trends in the data. (2) Creating Binary Features: for the *call type* feature, we create three binary features: CALL_TYPE_A, CALL_TYPE_B, and CALL_TYPE_C. These features will have a value of 1 if the corresponding *call type* is present and 0 otherwise. This encoding will allow the model to capture the

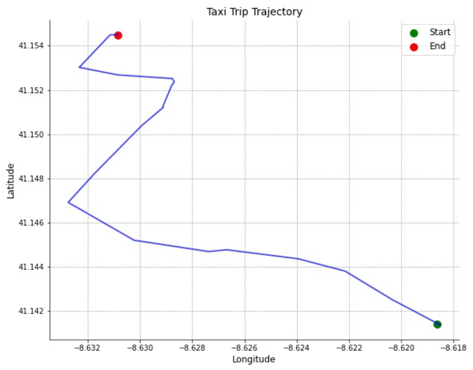

Figure 1: One sample data visualization in training data.

impact of different call types on the travel time. (3) Handling Missing Data: the *missing data* column indicates whether any GPS locations are missing. We create a binary feature called MISSING, where 1 represents missing data and 0 represents complete data. Since the test set only includes *Missing data* that is False, we instead filter out columns which have missing data. (4) Interaction Features: we create interaction features by combining multiple existing features. For example, we create a feature that represents the interaction between the *call type* and *timestamp* features. This can capture any combined effect of these two factors on the travel time. (5) Categorical Features: since features such as *origin call*, *origin stand*, *taxi id* represent categorical features, we re-encoded these features and made it categorical. (6) Dropping Irrelevant Features: if certain features are not expected to have a significant impact on the travel time or if they are not available during prediction, it might be beneficial to drop those features from the dataset. For example, if the *trip id*, *taxi id*, *day type* or *polyline* features are not expected to be useful, they can be removed. (7) data filetering: some data did not have an accurate travel time, which represents 0 or otherwise, we instead filtered out these rows to make a robust prediction. (8) Categorical encoding: the *origin call* feature identifies each unique customer in the dataset. This feature spans more than 50,000 unique values, presenting substantial variance. Certain customers have requested taxis only a couple of times, whereas others have done so more frequently. Specifically, customer number 2002 alone has requested over 55,000 taxi rides, accounting for nearly half the dataset's total requests. This rate of request, equivalent to 78 taxi rides per day, indicates that customer number 2002 is likely a public phone number shared among multiple users. We plan to incorporate *origin call* into our feature matrix, albeit with dimension reduction to only include customers who have requested taxis more than 100 times, reducing our dimensionality from 55,000 to just 253. (9) spatial features: we have a csv file which includes taxi stands location information as well as its latitude and longitude, we make the first row as the reference to calculate the The relative distance between stands as our continuous feature.

Finally, we apply StandardScaler() method to normalize our continuous data. StandardScaler scales the data to have zero mean (μ) and unit variance (σ). This transformation ensures that the data distribution remains centered around zero, maintaining the original shape of the distribution. It does not distort the relative distances between data points, preserving the overall structure of the data. Besides, StandardScaler is less sensitive to outliers compared to other normalization techniques[15]. Outliers can significantly impact the range and distribution of the data. By centering the data around zero and scaling it to unit variance, StandardScaler reduces the influence of outliers and helps in robust data analysis. Moreover, StandardScaler produces transformed data that is easily interpretable and comparable. The transformed features will have a similar scale, allowing for fair comparisons between them. This is particularly important when dealing with features measured in different units or with different scales.

$$x_{scaled} = \frac{(x_i - \mu)}{\sigma} \tag{2}$$

3 Deep Learning Model

After feature engineering and normalization processes, we got a feature that has 867 dimensions, we also applied MSE loss as our loss function. For the model architecture, we chose three deep learning models, multi-layer perceptron[5], TabNet[1] and TabTransformer[8]. The mathematical expression of the MSE is presented below, where n represents the batch size, y_i represents the ground truth label *travel time*, $\hat{y}$ represents the predicted output by deep learning models.

$$MSE = \frac{1}{n}\sum_{i=0}^{n}(y_i - \hat{y})^2 \tag{3}$$

3.1 Multi-layer perceptron

We design our first multi-layer perception model to predict travel time. The model is quite simple that it only includes six fully-connected layers and the activation function is LeakyReLU. We do not include any maxpooling, batch normalization, or dropout technique in this model, just in order to keep the model as simple as possible to become the baseline of the deep learning. The detailed model architecture is in Table 1.

Table 1: Multi-layer perception architecture

Architecture		
Name	Layer	Parameters
fc1	Linear(871,2048)	1.8M
fc2	Linear(2048,1024)	2.1M
fc3	Linear(1024,512)	0.5M
fc4	Linear(512,128)	65.7K
fc5	Linear(128,32)	4.1K
fc6	Linear(32,1)	33
Total		4.5M

3.2 TabNet

Recently, as the Transformer-based model performs the state-of-art results in various tasks such as natural language processing, computer vision, etc. It has also been proved to perform well in tabular data. The second model we apply is TabNet[1]. The model is a high-performance and interpretable canonical deep tabular data learning architecture, which uses sequential attention to choose which features to reason from at each decision step, enabling interpretability and more efficient learning as the learning capacity is used for the most salient features. In the TabNet architecture, the input features are initially transformed using a shared learnable embedding layer. This helps in capturing non-linear relationships and reducing the dimensionality of the input space. In feature selection process, TabNet uses a sequential decision-making process to select relevant features at each step. It employs adaptive feature selection masks that are learned during training. These masks determine which features are used for information propagation in subsequent layers. The selected features are then passed through multiple layers of the attentive transformer. This component applies self-attention mechanisms to capture important interactions and dependencies between the selected features. In feature aggregation process, at each layer, TabNet generates two outputs: the updated features and a set of feature importance scores. The feature importance scores represent the contribution of each feature towards the final predictions. The final layer of TabNet takes the output features and uses them to make predictions for the target variable. The architecture allows for various prediction tasks, such as regression or classification. Besides, batch-normalization and dropout techniques are applied in this model. The detailed architecture is listed in Figure 2.

3.3 TabTransformer

TabTransformer[8] is another Transformer-based deep learning architecture to predict tabular data results. In the original research paper, TabTransformer has been proved to perform well on many

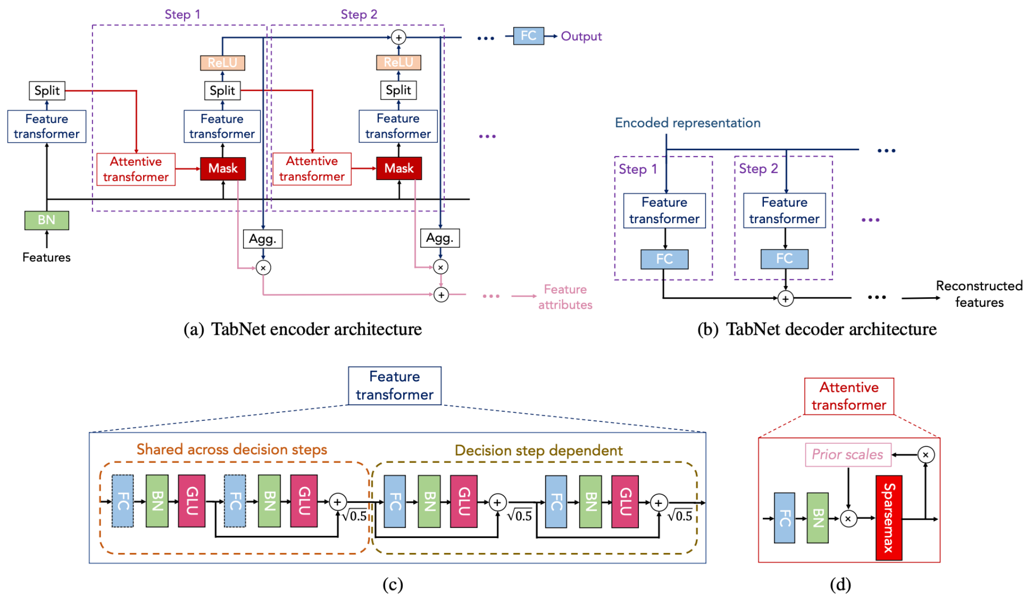

Figure 2: TabNet architecture developed by [1]

tasks compared with recent state-of-art machine learning methods such as XGBoost, LightGBM, CatBoost, etc. The TabTransformer is built upon self-attention based Transformers. The Transformer layers transform the embeddings of categorical features into robust contextual embeddings to achieve higher prediction accuracy. The TabTransformer architecture comprises a column embedding layer, a stack of N Transformer layers, and a multi-layer perceptron. Each Transformer layer consists of a multi-head self-attention layer followed by a position-wise feed-forward layer. The categorical features were encoded by *Column Embedding* method. The embeddings are learned in end-to-end supervised training using labeled examples. Layer normalization[2] and embedding dropout[7] techniques are applied in this architecture. The detailed architecture was presented in Figure 3.

3.4 Model summarize

In this task, we apply three types of deep learning models, one is multi-layer perception-based model, the other two are Transformer-based model. We did not include CNN or RNN-based model since research shows that these models performed well in computer vision and natural language processing tasks, but no more research shows that it outperforms well in tabular data tasks. We summarize each model's parameters and TFLOPs in Table 2.

Table 2: Summarize of the model architecture

Name	Architecture	Parameters (M)	TFLOPs(G)
MLP	MLP	4.5	9.16
TabNet	Transformer	1.75	12.62
TabTransformer	Transformer	4.4	124.47

4 Experiment Design and Results

Since the deep learning model especially Transformer-based model requires huge computational resources, we train our deep learning model by applying a remote deep learning server with Tesla A100 40G GPU and Intel Xeon 5260R CPU and 256G RAM. We apply PyTorch[14] and Sikit-learn[16] framework to train and test our model.

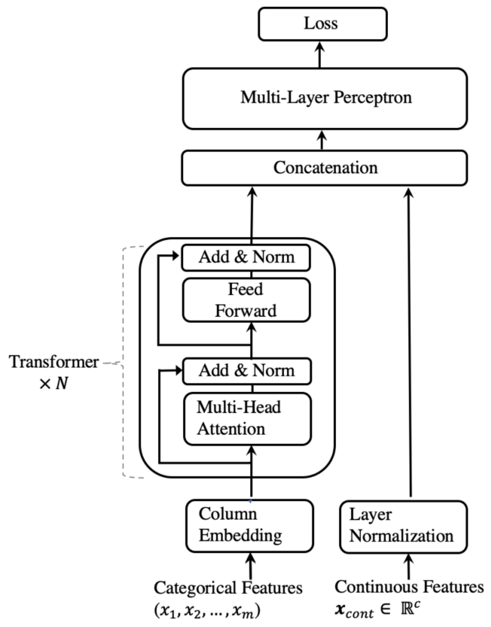

Figure 3: TabTransformer architecture developed by [8]

4.1 Hyperparameters and evaluation

To make a reasonable comparsion, in addition to applying deep learning model such as MLP, TabNet and TabTransformer, we also used several state-of-art GBDT-based machine learning model as our baseline, XGBoost[4] and LightGBM[9], which has been proved to outperform well on tabular data compared with some deep learning models.

The optimizer we used to train the deep learning model was Adam, with a weight decay of $5e-4$, batch size for the training and validation data was 2048, we decayed the learning rate of each parameter group by $\gamma = 0.92$ every 50 epochs. Because we have enough computational resources, we set the batch size a little larger to speed up the training process. Then, we trained every deep learning models for at most 200 epochs, and in order to reduce the problem of over-fitting, we set an early stopping rule that if the model did not promote the performance in validation set within 10 epochs, we terminated the model training process early. Next, we set the learning rate for MLP is $1e-3$, for TabNet is $2e-2$, and for TabTransformer is $5e-2$. In Figure 4, we compared different learning rate choices for the three models. The performance is evaluated on the validation dataset. The learning rate we chosed was the most suitable one to predict the model's output. Besides, for machine learning models, we set $\epsilon = 0.01$ for XGBoost and $lr = 0.3$ for LightGBM.

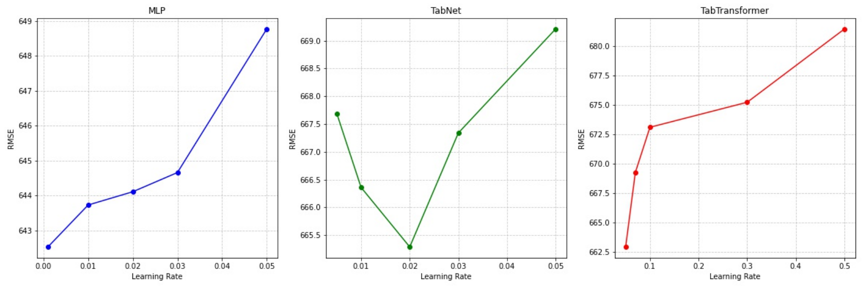

Figure 4: Adjustment of the deep learning model's learning rate.

Just as most regression tasks process did, we evaluate our performance on the validation set by using RMSE error. Lower RMSE values suggest better model performance.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=0}^{n}(y_i - \hat{y})^2} \tag{4}$$

4.2 Model performance

To evaluate model performance, we firstly downloaded the public test set from Kaggle, with 320 rows and 8 columns. In order to predict these results by applying our well-trained deep learning models, we firstly reapplied the same feature engineering and normalization techniques to generate new features. Then, we applied our models to predict the results and finally uploaded to the kaggle competition platform. The RMSE scores for the test set is our final evaluation indices for each model. We also calculated the model's training time for each training process. The performance results are listed in Table 3. Since we uploaded several times, we only report the best performance for each model. To speed up training process, we applied one Tesla A100 GPU to train the deep learning models, including MLP, TabNet and TabTransformer. For machine learning models, we used CPU with 8 threads to speed up the process.

Table 3: Model performance in the public test set

Name	RMSE	Parameters (M)	TFLOPs(G)	Training time(s)
MLP	770.33	16.8	26.75	28.44
TabNet	771.35	1.75	12.62	106
TabTransformer	739.57	4.4	124.47	240
LightGBM	758.06	/	/	3.42
XGBoost	**732.37**	/	/	**3.13**

The performance results showed that the machine learning model XGBoost outperforms other models in this regression task, as well as it requires less training time compared with deep learning models. Transformer-based models, on the other hand, reached a comparable results, for example, models like TabTransformer got a 739.57 RMSE score which was almost comparable with XGBoost model. However, the drawback of these Transformer-based model is also apparent, which may require much more resources and training time to reach the convergence point of the model.

4.3 Over-fitting issues

Over-fitting is a common issue in machine learning where a model performs exceptionally well on the training data but fails to generalize well to unseen or new data[19]. It occurs when a model becomes too complex or overly specialized to the training data, capturing the noise or random fluctuations in the data rather than the underlying patterns or relationships. We applied several techniques to avoid the issue of over-fitting, such as dropout, batch normalization, layer normalization, L2 regularization, etc. However, some deep learning models, especially the designed MLP model, the over-fitting problem is the most serious.

In Figure 5, we plot the average training loss and validation loss for each model, for MLP, it showed that the training loss was declined while the validation loss did not change a lot in the beginning and increase during the training process, which indicated that the MLP model was suffered from an over-fitting issue. TabNet also had the same issue just as MLP. But fortunately, we did not find a similar pattern in TabTransformer, it might because these models were more complex and apply several normalization and dropout techniques. Therefore, to address this issue, we only saved the best model before the validation loss was increasing.

4.4 Results visualization

TabTransformer performs the best results compared with other two deep learning model, therefore, we visualized the predicted results from the well-trained TabTransformer in Figure 6. Our model ranks 35 in the competition and the final test RMSE is 732.37.

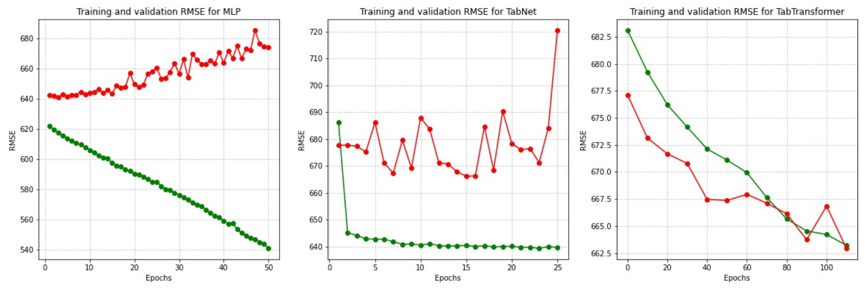

Figure 5: Over-fitting issues for deep learning models. (Red line indicates the training RMSE, while the green line indicates the validation RMSE.)

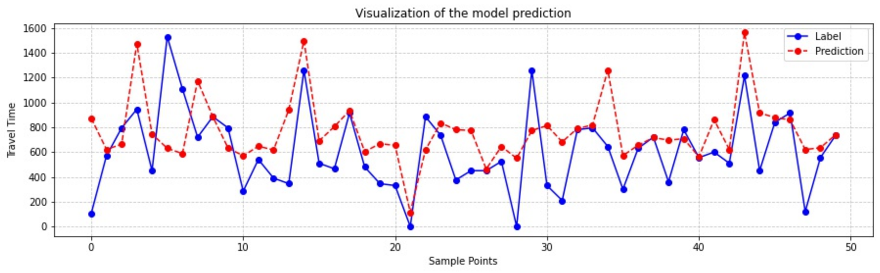

Figure 6: Model prediction results for TabTransformer. (Every point in the x-axis represents one sample.)

5 Discussion and Future Work

In this work, we build a regression model to predict the taxi travel time in Portugal and achieve a comparable result in the Kaggle competition. However, beyond this task, there are still many research to conduct in order to improve the model performance and increase the generalizability of the deep learning model. The most effective feature engineering strategy is building interaction features. Because the provided dataset only has 8 raw features, which was lack of intractability. Building interaction features can provide a more expressive representation of the data. By combining multiple features, we can create more informative and discriminative representations that improve the model's ability to separate different classes or make accurate predictions. In addition, building interaction features can lead to improved model performance by providing additional information and capturing hidden patterns in the data. It can help the model overcome limitations of individual features and enhance its ability to generalize and make accurate predictions. Moreover, many real-world problems have non-linear relationships between variables. By creating interaction features, we can capture these non-linearities that may not be adequately represented by individual features. This allows the model to learn more complex relationships and improve its predictive power.

In our experiment, we found the most effective method to improve the model performance and competition ranking is hyper-parameter tuning. To be honest, there is no significant difference between each model, however, better hyper-parameter tuning techniques would lead to better performance. We applied grid search method to find the best learning rate for our model, which shows a significant improvement to our final prediction results.

The biggest bottleneck for this project would be the dataset itself. Just as we mentioned above, the dataset only includes 8 raw features, every feature is category variables, and there is no continuous features in this dataset except for the target label *polyline*. To reach a better performance for this task, we have to conduct many complex feature engineering tasks to construct various features, as well as efficiently encode the categorical variables. Another issue is that the target variable *polyline* provided a lot of hidden informations, such as total distance, elucidian distance, number of turns, city locations, etc. Since the test set did not contain such *polyline* information, we could not apply those information to our predicted model. To improve the model performance, better feature engineering method should be included.

For beginners in relevant deep learning tasks, we have several suggestions. First, collect the relevant data for the prediction task and preprocess it appropriately. This may involve handling missing values, scaling numerical features, encoding categorical variables, and splitting the data into training and validation sets. Ensure that your data is representative and properly formatted for training a deep learning model. Second, start with simpler models: As a beginner, it's advisable to start with simpler models before diving into complex architectures. Begin with basic neural network models like feed-forward neural networks or simple convolutional neural networks (CNNs). This will allow you to grasp the fundamental concepts and get a hands-on understanding of model design, training, and evaluation. Third, learn from existing architectures: Explore and study existing deep learning architectures that have been successful in similar prediction tasks. Familiarize yourself with popular models like recurrent neural networks (RNNs), long short-term memory (LSTM) networks, and transformer models. Understand their architectural components, such as layers, activation functions, and regularization techniques. Fourth, experiment with model architectures: Once you have a solid foundation, start experimenting with different model architectures to improve performance. You can try adding additional layers, increasing network depth, or exploring different activation functions and regularization techniques. Keep track of your experiments and compare the results to understand the impact of different architectural choices. In addition, optimize hyperparameters: Deep learning models have various hyperparameters that control the model's behavior. Experiment with different hyperparameter settings such as learning rate, batch size, optimizer, and regularization strength to find the optimal combination that maximizes performance on your validation set. Techniques like grid search or random search can help you explore different hyperparameter configurations efficiently. Moreover, regularly evaluate and analyze results: Evaluate your model's performance using appropriate evaluation metrics for your prediction task. Monitor both training and validation performance to identify potential issues like overfitting or underfitting. Analyze the model's predictions, visualize the results, and interpret any patterns or insights gained from the model's behavior. Also, don't forget to learn from the community: Deep learning is a rapidly evolving field with a vast community of researchers and practitioners. Engage in online forums, read research papers, and join communities to learn from others, ask questions, and stay updated on the latest advancements. Platforms like GitHub, Kaggle, and online courses provide valuable resources and opportunities to collaborate with fellow learners.

There are many ideas that we could explore if we have more time and resources. First, in section 4, we found that the machine learning model such as XGBoost outperformed all of the deep learning models. It proved that although many research showed that some deep learning architecture performed well in tabular data, several machine learning model still lead in tabular data analysis tasks. Since tabular data is low-dimensional compared with images and texts, deep learning techniques did not perform well on these types of tasks. However, with the development of many big generalizable models, such as Large Language Model (LLM) like GPT-3[3], segmentation model like SAM[10], there is much work to do to develop a well-performed deep learning model to adjust the low-dimension nature of tabular data. Second, since different datasets require different model architectures, it is much harder for us to design a proper deep learning model. However, there is one technique called Nerual Architecture Search (NAS)[20], which aims to find optimal or highly effective architectures for specific tasks without relying on manual design or human expertise. Therefore, if we have more resources, we could apply NAS to design the deep learning model to reach the best performance. Third, large scale per-training is another effective technique to improve the results[17]. Pre-training deep learning models on large-scale datasets, such as unsupervised or self-supervised learning, has shown great potential in improving generalization and transfer learning capabilities. With more resources, we would invest in exploring and refining pre-training techniques on even larger datasets, potentially leveraging unlabeled data from the web or specialized domains. Finally, explainability and interpretability is another interesting area that we would like to explore[6]. Deep learning models often lack interpretability, making it challenging to understand their decision-making process. Investing in research to improve the interpretability and explainability of deep learning models would be valuable. Techniques such as attention mechanisms, visualization methods, and model-agnostic interpretability approaches could be explored to shed light on the inner workings of complex models.

References

[1] S. Ö. Arik and T. Pfister. Tabnet: Attentive interpretable tabular learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 6679–6687, 2021.

[2] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[3] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[4] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[5] M. W. Gardner and S. Dorling. Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. *Atmospheric environment*, 32(14-15):2627–2636, 1998.

[6] D. Gunning and D. Aha. Darpa's explainable artificial intelligence (xai) program. *AI magazine*, 40(2):44–58, 2019.

[7] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[8] X. Huang, A. Khetan, M. Cvitkovic, and Z. Karnin. Tabtransformer: Tabular data modeling using contextual embeddings. *arXiv preprint arXiv:2012.06678*, 2020.

[9] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 30, 2017.

[10] A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo, et al. Segment anything. *arXiv preprint arXiv:2304.02643*, 2023.

[11] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[12] Y. Lv, Y. Duan, W. Kang, Z. Li, and F.-Y. Wang. Traffic flow prediction with big data: A deep learning approach. *IEEE Transactions on Intelligent Transportation Systems*, 16(2):865–873, 2014.

[13] A. Miglani and N. Kumar. Deep learning models for traffic flow prediction in autonomous vehicles: A review, solutions, and challenges. *Vehicular Communications*, 20:100184, 2019.

[14] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[15] S. Patro and K. K. Sahu. Normalization: A preprocessing stage. *arXiv preprint arXiv:1503.06462*, 2015.

[16] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[17] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[18] Y. Wang, D. Zhang, Y. Liu, B. Dai, and L. H. Lee. Enhancing transportation systems via deep learning: A survey. *Transportation research part C: emerging technologies*, 99:144–163, 2019.

[19] X. Ying. An overview of overfitting and its solutions. In *Journal of physics: Conference series*, volume 1168, page 022022. IOP Publishing, 2019.

[20] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.