CSE 251B Project Final Report

https://github.com/kennethvuongcode/UCSD-CSE-251B-Class-Competition-2025

Andrew Xi
anxi@ucsd.edu

Kenneth Vuong
k5vuong@ucsd.edu

Zerlina Lai
z6lai@ucsd.edu

Abstract

This project tackles the problem of forecasting the future path of a vehicle in complex traffic scenarios using a deep learning approach. We evaluate several baseline models, including constant velocity, MLP, and LSTM, and propose a hybrid Transformer-LSTM architecture that captures both spatial and temporal dependencies. Our model incorporates design choices such as sine/cosine heading representation and context injection of final position and velocity. We also conduct ablation studies to isolate the impact of various architectural and preprocessing decisions. Our final model scores an MSE of 9.95966 on the private leaderboard.

1 Introduction

The goal of this project is to predict the future trajectory of a specific vehicle (the ego-vehicle) using its motion history and surrounding context in real-world driving scenes. This deep learning task is critical for ensuring safe and efficient autonomous driving. Accurate trajectory prediction allows autonomous systems to anticipate and respond to the actions of others on the road, preventing collisions, creating optimal routes, and enabling real-time decisions. For example, predicting pedestrian crossings or the motions of other vehicles during unprotected left turns can reduce fatal accidents that even human drivers commonly make.

The starter code provides a foundational framework for loading data, pre-processing inputs, and evaluating trajectory predictions using two simple baseline models: a constant velocity model and a multilayer perceptron (MLP) model.

The constant velocity model assumes that the ego-vehicle will continue moving at a constant velocity in a straight line. Instead of using the actual measured velocities in the x and y directions, the velocity is calculated as the difference in position between consecutive time steps. The average velocity is computed over the past 50 time steps and used to linearly project the ego-vehicle's position over the next 60 time steps. While this approach is simple and straightforward, it has several limitations. It does not account for changes in the ego-vehicle's movement or the behavior of other agents. It also predicts only a linear trajectory from the last time step, preventing it from handling turns, stops, and accelerations. As a result, it provides a useful baseline to benchmark complex models, but its predictions tend to be inaccurate for dynamic trajectories, making it dangerous to deploy in real-world environments.

On the other hand, the MLP served as a learning-based baseline for this task. The model takes the past 50 time steps (5 seconds) of every agents' motion history and flattens them into a single vector for input. Its architecture consists of four fully connected layers with 1024, 512, 256, and 120 output neurons, corresponding to the (x,y) coordinates for the future 60 time steps of the ego-vehicle. ReLU activations and dropout regularization are implemented in between layers. The model is trained to

Preprint. Under review.

minimize Mean Squared Error (MSE) loss between the predicted and ground-truth positions of the ego-vehicle and is optimized using the Adam optimizer. During inference, test scenes are passed through the trained MLP to produce predicted coordinates. Some limitations of this model include a lack of spatial and temporal awareness due to flattening the input and overparameterization due to lack of data for such a large network.

Our final solution, a hybrid Transformer-LSTM trained to minimize MSE, improved on the results shown by the MLP. The Transformer-LSTM was able to learn temporal and spatial dependencies to a greater extent, improving the model's accuracy beyond that of any previous solution we implemented. However, this model still has much room for improvement in learning key spatial and temporal features that impact the ego agent's trajectory.

2 Related Work (optional)

3 Problem Statement

This project addresses the problem of motion forecasting in the context of autonomous driving. Specifically, the goal is to predict the future trajectory of the ego-vehicle in complex, dynamic, real-world driving scenes involving many outside agents, such as cars, cycles, and pedestrians.

We use a modified version of the Argoverse 2 dataset, which consists of 11-second driving scenarios. Each scenario captures the scene in a 2D bird's eye view of the environment, with agent behaviors sampled at 10 Hz (10 time steps per second). These scenarios feature diverse agent behaviors, complex social interactions, and rare events, making long-range prediction particularly challenging and realistic for autonomous systems. This is an especially crucial and challenging task, as even half a second can change an outcome in the real world.

Each driving scenario is represented as a tensor with shape $X \in \mathbb{R}^{A \times T_{total} \times d}$, where A = 50 is the number of agents, T_{total} = 110 is the total number of time steps (corresponding to 11 seconds), and d = 6 is the number of features per agent per time step. These features include the agent's position in the x direction (position_x), position in the y direction (position_y), velocity in the x direction (velocity_x), velocity in the y direction (velocity_y), heading angle (heading), and object type (object_type). The full dataset can be represented as a tensor $X \in \mathbb{R}^{N \times A \times T_{total} \times d}$, where N is the number of scenarios.

For training, we use 10,000 driving scenarios. The input to our model is the first 50 time steps (representing 5 seconds) of each scene, resulting in an input tensor of shape $X \in \mathbb{R}^{10,000 \times 50 \times 50 \times 6}$. Each input provides information on the entire scene across all agents. The model is trained to predict the future trajectory of the ego-vehicle over the next 60 time steps (6 seconds), producing an output tensor of shape $X \in \mathbb{R}^{600,000 \times 2}$. Each row corresponds to the ego-vehicle's predicted (x, y) position for one future time step in a scenario (10,000 scenarios and 60 time steps resulting in 600,000 rows).

During evaluation, we use a held-out set of 2,100 unseen scenarios. The input tensor has the same format as the input used during training; it captures the behavior of all agents in the first 50 seconds of each scenario. This results in an input tensor of shape $X \in \mathbb{R}^{2,100 \times 50 \times 50 \times 6}$. The output predictions are structured as a tensor of shape $X \in \mathbb{R}^{12,600 \times 2}$. Each row corresponds to the ego-vehicle's predicted (x, y) position for one future time step in a scenario (2,100 scenarios and 60 time steps resulting in 12,600 rows).

We preprocessed the raw input data to allow the model to better learn patterns from it, adding two features for a total of 8 (see Methods section). We batched data with batch size 32, resulting in training batches of shape 32x50x50x8. To train our model, we reserve 90% of the data for training and leave the remaining 10% for validation. Validation loss is a crucial step in quantifying how well our model learns, as well as a useful metric to decide when to stop training.

4 Methods

4.1 Data Preparation

We create feature vector inputs by preprocessing the raw input data to allow the model to better learn patterns. First, we normalize the large position and velocity values to a range that Transformer

architectures are better suited for– they perform poorly on data on large numerical scales. We scale the position and velocity x,y data down by a factor of 7.

Instead of using raw x,y position coordinates, we set the origin to the position of the ego vehicle at the last training timestep. This allows the model to be invariant to global position in the map, only learning relative motion patterns, which are more general and reusable.

In addition, we represent heading in raw radians in the range $[+\pi, -\pi]$ as well as sine and cosine form, to allow a richer angular representation. It helps the model learn that heading is cyclical, where $+\pi$ and $-\pi$ are identical. This step adds two additional features sin(heading) and cosine(heading) to the data vector for each timestep.

We also augment the dataset by randomly applying rotation and mirroring to scenes with a 50% chance.

We batched data with batch size 32, resulting in training batches of shape 32x50x50x8. The 8 features are position x, position y, velocity x, velocity y, raw heading, object type, sin(heading), and cos(heading). To train our model, we split 90% of the data for training and the remaining 10% for validation.

4.2 Best Model Architecture

Our best performing model was a hybrid LSTM/Transformer (Figure 1).

Our encoder is designed to process trajectory sequences for multiple agents using a Transformer-based architecture. First, we project the input features from the input 8 dimensions to 128 hidden dimensions. Since transformers are not intrinsically order-aware, positional encoding is added. We use sinusoidal positional embeddings because our input data consists of temporally equally spaced snapshots of a scene. Therefore a fixed position encoding using sine and cosine functions (as in the original Transformer paper [1]) is a good fit for the task.

The encoder contains three layers. Each layer uses 8-head self attention to model dependencies across time steps. A feedforward network, layer normalization, and dropout are also included in each layer. The flow of data through the encoder is as follows: raw inputs are converted to embeddings, positional encoding is added, and the result is passed through the multi-layer transformer. Then, the output from the last timestep is used as the agent's encoded representation. Lastly, we normalize the final embedding to stabilize training.

Before passing the encoded representation to the decoder stack, we add explicit context by concatenating the ego agent's last known position and velocity to it.

Our decoder is a LSTM-based decoder that takes a latent vector from the encoder and the position and velocity context, then generates a predicted future trajectory as a sequence of 60 2D positions over time.

The LSTM takes in the augmented latent vector for each timestep. It then learns temporal dynamics to generate a meaningful hidden state evolving over 60 future timesteps. A MLP maps the LSTM output at each timestep into a delta x, y offset, predicting step-wise relative movement rather than the absolute positions.

Lastly, we take the cumulative sum of the predicted relative movement to output x,y positions. We denormalize the predicted coordinates by shifting the origin back to its original position then scaling up by a factor of 7. The output is then a series of x,y positional coordinate predictions over 60 future timesteps.

4.3 Training Procedure

We implement early stopping based on validation loss with a patience of 5 epochs. If the validation loss does not improve for 5 consecutive epochs, then the model is saved and training stops. We use MSE as the loss metric evaluated on the normalized data, and Adam optimizer with learning rate set to 5e-4.

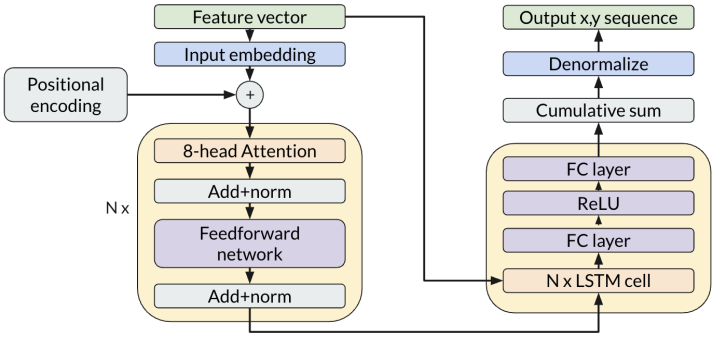

Figure 1: Diagram of our Custom LSTM/Transformer

4.4 Key Contributions

The key contributions and insights from this final solution are:

1. Data setup: using sine/cosine to represent heading direction rather than angle in radians allows the model to better handle rotational information.
2. Directly injecting context to the latent vector before decoding significantly improved the performance of the model compared to the same architecture without injection. The improvement in terms of minimum MSE loss, within 40 training epochs, was roughly 1. This provides insight to the importance of context in helping models learn quickly and well.
3. More complex models do not necessarily mean better performance. With the right hyperparameter tuning, even a simple LSTM-based model can have high performance on sequential prediction tasks.

4.5 Future Improvements

Although this model is able to learn many of the nuances of trajectory prediction, there are several limitations that constrain its performance.

1. Loss of early temporal context:
   The decoder relies on the final hidden state of the encoder, which compresses the trajectory history into one vector. This bottleneck can lead to the loss of valuable information from earlier timesteps, especially in long sequences or scenes with complex temporal dependencies.
2. Limited attention over scene context:
   Although the model uses multi-head self attention in the encoder, it is not structured to distinguish between temporal and inter-agent interactions. There is no specialization of attention along temporal or agent dimensions, which may limit the model's ability to focus on specific impactful interactions. A more targeted attention mechanism with separate temporal and cross-agent attention blocks could help the model better capture structured dependencies in the scene.

5 Experiments

5.1 Baselines

1. **Constant velocity model**. This simple, rule-based model was provided by the starter code. It assumes that the ego-vehicle will continue moving at constant velocity and direction starting from the last observed timestep in the input sequence. First, the difference is taken between both the x and y positions between consecutive time steps and then averaged to obtain an average velocity in both the x and y direction. Future positions (next 60 time steps) are predicted by linearly extrapolating the final position using this equation:
   $\hat{p}_t = p_{49} + (t+1)\bar{v}$ for $t \in [0, 59]$

This model's inputs only require the x and y positions of the ego-vehicle and involves no learned parameters. It obtained a test MSE of 53.02926 on the Kaggle leaderboard.

2. **LSTM (Long Short-Term Memory) Network.** Another baseline model provided by the starter code. This is a learning-based baseline using a recurrent neural network architecture designed to capture temporal dependencies in the input sequence. Specifically, a single-layer LSTM is implemented which takes the historical trajectory of the ego-vehicle as input. The input to the LSTM consists of a sequence of 50 time steps, where each step contains a 6-dimensional feature vector (e.g., position, velocity, and heading-related information). This results in an input tensor of shape $(batchsize, 50, 6)$. The LSTM processes this sequence and produces a hidden state at each time step; we extract the hidden state from the final time step and pass it through a fully connected (linear) layer to predict the future trajectory.

   The network has a hidden dimension of 128 and an output dimension of 120 (corresponding to 60 future time steps in 2D space). Training is performed using MSE loss and the Adam optimizer with a learning rate of 1e-3. This model captures temporal structure in the ego-vehicle's motion more effectively than feedforward baselines. This model obtained a test MSE of 10.45115 on the Kaggle leaderboard.

3. **Multilayer Perceptron**. This is another model that was provided by the starter code and serves as a learning-based baseline for this task. The MLP is a fully connected feedforward neural network that takes the flattened history of all 50 agents over 50 time steps as input (5000 input features) to predict the ego-vehicle's future trajectory (120 output features). The network has 4 fully connected layers with hidden dimensions of 1024, 512, 256, and 120, respectively. Each layer is accompanied by a ReLU activation and Dropout for nonlinearity and regularization. In total, the model has approximately 6 million parameters. Training was conducted using MSE loss and the Adam optimizer with a learning rate of 1e-3, for 10 epochs and a batch size of 64. This model obtained a test MSE of 57737.87279 on the Kaggle leaderboard.

4. **Linear Regression**. This is a simple and interpretable model that served as an early baseline for this task. Instead of working with only the default, raw features, we implemented a feature engineering pipeline that extracted features such as positions, velocity, and headings at regular intervals, total distance traveled, distance to nearby agents, and more. We manually experimented with different combinations of these features using cross-validation to find the best subset. Parameters involve the weights for each feature. The final features were flattened and inputted into a linear model:

   $\hat{y} = \beta_0 + \sum_{i=1}^{d} \beta_i x_i$

   This model was implemented using the sklearn library. In the end, the best model used the following features: the ego-vehicle's position, velocity, and heading every 10 time steps (including the last time step), its average velocity and heading, and its object type. This obtained a test MSE of 13.00014 on the Kaggle leaderboard.

5.2 Exploratory Models

1. **Lasso Regression.** To address overfitting issues with Linear Regression, we experimented with Lasso regularization, which adds an L1 penalty term to prevent feature weights from getting too large. This encourages sparse weights with the following objective:

   $\arg\min_w ||\hat{y} - y||_2^2 + \lambda ||w||_1$

   This effectively performs some feature selection by setting the weights of unimportant features to 0. Hyperparameters involved regularization strength λ, which was tuned using cross-validation. In the end, the best Lasso Regression model used a $\lambda = 0.1$ to obtain a test MSE of 11.82991 on the Kaggle leaderboard (all engineered features were fed in as input due to its implicit feature selection).

2. **Ridge Regression.** In addition to Lasso Regression, we also worked with Ridge Regression, which introduces an L2 penalty to shrink weights towards zero without eliminating them. This method reduces model complexity and improves generalization. The objective function is:

   $\arg\min_w ||\hat{y} - y||_2^2 + \lambda ||w||_2^2$

Like Lasso Regression, λ was tuned using cross-validation. The best Ridge Regression model used the same features as the Linear Regression model and a $\lambda = 0.1$ to obtain a test MSE of 11.97326 on the Kaggle leaderboard.

3. **Basic Transformer Implementation** In an effort to capture temporal and spatial dependencies, we design a Transformer-based model to predict the trajectory of the ego-vehicle. In which, we embed and encode the input features, then implement a Transformer encoder, and decode the final output of the encoder with an MLP.
   Specifically, the model takes as input a sequence of 50 time steps of the ego-vehicle's 6-dimensional features. The input features are first projected into a higher-dimensional embedding space (256 dimensions) via a linear layer. Learnable positional encodings are added to the embedded input to inject temporal order information into the model. The resulting sequence is then passed through a stack of 4 Transformer encoder layers, each with 4 attention heads and a feedforward dimension of 256. A fully connected MLP decodes the final encoder output (after flattening across the time dimension) into the predicted future trajectory of the ego-vehicle across 60 time steps.

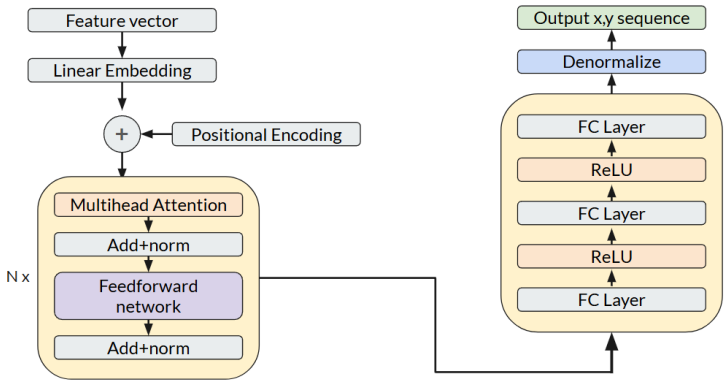

Figure 2: Diagram of Basic Tranformer Encoder Model

The model was trained using the Adam optimizer and mean squared error loss. Dropout with a rate of 0.2 was applied in the Transformer layers to reduce overfitting. This Transformer implementation achieved a test MSE of 12.53511 on the Kaggle leaderboard. This high MSE score is likely due to only the ego-agent information being considered in the scene. If the entire scene of all agent data was considered, the Transformer would likely perform much better.

5.3 Evaluation

We evaluate our models and baselines using Mean Squared Error (MSE) between the predicted and ground truth trajectories of the ego-vehicle. This metric is calculated over the 60 future time steps, comparing the predicted (x,y) positions to the true positions. MSE provides a straightforward and interpretable measure of prediction accuracy in terms of how far the predicted values deviate from the actual value. We compute the MSE with the following equation:

$$MSE = \frac{1}{N \times T_{pred}} \sum_{i=1}^{N} \sum_{t=1}^{T_{pred}} (y_{i,t} - \hat{y}_{i,t})^2$$

where N is the number of scenarios and $T_{pred} = 60$ is the number of future time steps. This helps capture prediction error across all positions and time steps in every scenario.

Additionally, the Kaggle public leaderboard calculates the MSE using the test set to evaluate generalizability. No other custom metrics were used or implemented in this work.

5.4 Implementation details

In order to train on larger GPUs, we opted to utilize cloud services as our personal computers did not have advanced GPUs. Initially, we utilized Google Colab's T4 GPUs, however we quickly reached their free usage limit. For our final model, we trained and ran inference on Kaggle for their longer

GPU usage limit, utilizing their NVidia K80 GPUs. It takes multiple minutes to train our model for one epoch going through the entire training data set once, with batches of size 32. We chose our initial hyperparameters as common settings from our previous work, such as learning rate 1e-5, hidden dimension 128, 4 heads for attention, 2 transformer layers, etc. Then we fine tuned them by increasing and decreasing the model dimensions and number of heads by factors of 2, trying 1 and 3 transformer layers, and using learning rate 5e-4 and 1e-4. We found that combining hidden dimension above 256 with 8 attention heads and 4 transformer layers trained too slowly to be able to evaluate results within a desired time frame, so that was the upper cap for our fine tuning experiments. In the end, we achieved the best performance within a reasonable time frame with hidden dimension 128, 8 attention heads, 3 transformer layers, 0.1 dropout, and Adam optimizer with learning rate 5e-4.

5.5 Results

The following are the results from the Kaggle leaderboard.

Model	Private	Public
Constant Velocity	53.02926	53.16972
LSTM	10.37899	10.45115
Baseline MLP	57598.64530	57737.87279
Linear Regression	13.19137	13.00014
Ridge Regression	12.13636	11.97326
Lasso Regression	11.94206	11.82991
Basic Transformer Encoder	12.60749	12.53511
Custom LSTM/Transformer	9.95966	9.90878

Table 1: Model Test MSE Results

5.6 Ablations

1. **Context Injection** In our full model, we concatenate the ego-vehicle's final observed position and velocity to the encoded latent representation before passing it to the LSTM decoder. To evaluate the effect of this addition, we trained an identical architecture without the added context injection. Removing the context led to a significant increase in validation loss (approximately 0.7 MSE), suggesting that providing short-term history at decoding time enables better prediction accuracy.
2. **Positional Embedding**

   We tested the model with learned positional embeddings vs. fixed sinusoidal positional embeddings in the Transformer encoder. Replacing learned embeddings with fixed positional embeddings led to a very slight improvement in MSE, small enough that it was unclear whether the difference was due to chance or a genuine performance gain. In either case, we kept the positional embeddings as they are simpler.
3. **Encoder Choice: Transformer vs. LSTM**

   As a control experiment, we used a LSTM as the encoder instead of the Transformer. The performance dropped sharply: test MSE increased by about 1.5 points. This shows the transformer encoder is significantly better at learning sequential dependencies due to attention mechanisms than the LSTM.
4. **Decoder Choice: MLP vs. LSTM**

   As another experiment, we replaced the LSTM decoder with a two-layer MLP. The performance decreased similarly to what we saw when simplifying the encoder. This result supports the LSTM's advantage in modeling sequential dependencies across the prediction steps compared to a simple feedforward decoder.

6 Conclusion

In this project, we tackled the problem of long-term trajectory prediction for the ego-vehicle in complex multi-agent driving environments. Starting from the simple baselines provided by the starter code, including constant velocity, LSTM, MLP, and Linear Regression models. We explored a wide

range of model architectures and data representations to improve performance on this forecasting task, including Lasso Regression, Ridge Regression, and a basic transformer architecture.

Our final solution was a hybrid LSTM/Transformer architecture. It achieved our best score on the Kaggle test leaderboard, achieving a public leaderboard MSE of 9.91 and private leaderboard MSE of 9.96, outperforming all other baselines and exploratory models. This architecture leveraged the temporal modeling capabilities of Transformers in the encoder, combined with the sequential generation strength of LSTMs in the decoder. It also integrated key design improvements, such as heading represented using sine and cosine, normalization with respect to the ego-vehicle's origin, and explicit inclusion of final position and velocity context.

Key Lessons and Insights

- Temporal encoding matters: Positional encoding and relative representation of the scene enabled the model to focus on motion patterns rather than absolute positions, leading to better generalization.
- Simplicity can be efficient: Despite testing larger Transformer-only models, the final architecture's performance shows that combining simpler components (e.g., a small LSTM decoder) with careful data handling can yield competitive results.
- Contextual information is powerful**:** Injecting final known position and velocity into the latent code reduced loss, highlighting the importance of conditioning on recent history.

Limitations

- Our model only used ego-agent information in many exploratory models. While the final architecture incorporated multiple agents, the decoder still relied on a compressed latent context, possibly discarding early temporal cues.
- We did not experiment extensively with explicit interaction modeling between agents, which could further improve predictions in crowded or multi-modal scenarios.
- Due to time and computational constraints, full hyperparameter tuning was limited, and training stability could be improved with more compute and longer runs.

Future Work

- Implement a full Transformer encoder decoder architecture with multi-head self attention that distinguishes between temporal and inter-agent interactions. This should incorporate all agents as input to a spatiotemporal encoder could allow the model to learn richer social dynamics.
- Explore other architectures and embedding approaches which might further incorporate context like agent proximity or inter-agent velocity.

For deep learning beginners working on similar sequential prediction tasks, we recommend starting with simple, interpretable baselines such as linear models or LSTMs before jumping into more complex architectures like Transformers. They should first build intuition around how the model handles time-series data and how preprocessing impacts learning. Normalize inputs, especially position and velocity, and represent angular data using sine and cosine to handle circularity. Avoid flattening time-series data unless necessary to conserve temporal structure. When testing, start small, overfit on a single batch to debug, and incrementally scale up. Also, visualizing predictions is incredibly helpful for identifying failure cases. Finally, prioritize learning by experimenting with architectural components (e.g., encoders, decoders, attention blocks) and training strategies (e.g., loss functions, optimizers) one at a time to understand how each part affects the model as a whole.

Ultimately, this project highlighted the importance of balancing architectural complexity with data-aware preprocessing and simple but effective design choices. Our model demonstrates that carefully structured deep learning models can achieve accurate, generalizable predictions for real-world autonomous driving scenarios.

7 Contributions

- Zerlina Lai: experimented with various LSTM/Transformer architectures, including producing and fine tuning the final model. Wrote parts of milestone, slides, and final report.
- Andrew Xi: experimented with baselines, Linear, Ridge, and Lasso Regression. wrote parts of milestone, slides, and final report
- Kenneth Vuong: experimented with baselines, exploratory LSTM and exploratory Transformer architectures. Wrote parts of milestone, slides, and final report.

8 References

References

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.