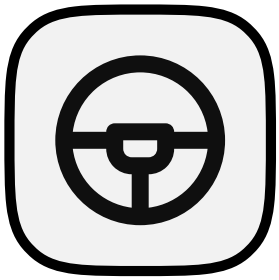

Efficient temporal awareness for autonomous vehicles using State Space Models

Student, 11th grade
Asandei Ștefan-Alexandru

Supervisor teacher
Pădurariu Emanuela-Tatiana

Romanian Science and Engineering Fair

October 16, 2024

Abstract

For the last couple of years, the attention mechanism has become the dominant method for developing advanced models. Transformer-based models have proven very capable in understanding language and vision [1], though coming at the cost of their quadratic complexity in scaling. A new architecture based on state space models, Mamba [2], addresses these concerns, however, having a lack of experimental backing. Therefore, this work presents a Mamba-based model, which proves to be very effective in the task of short-term video understanding, in the context of autonomous vehicles. Our end-to-end network predicts the vehicle speed, steering wheel angle, and future trajectory, three correlated features which the model seemed to understand their behavior. Secondly, we also trained two classic models on the same dataset for benchmarks. We show that it can outperform foundational models while also being faster at inference. Afterward, we also release training and inference code, as well as pre-trained model weights [1] to serve as educational resources for developers interested in Mamba models.

[1]Inference code and pre-trained models are available at `https://github.com/stefanasandei/steer`

Contents

1. Introduction

The task of processing sequenced data has emerged as a crucial capability for deep learning models. The use cases of this lead to language, time series forecasting, image, and video understanding models. Probably, one of the most well-known generative models is GPT-4 by OpenAI, used in ChatGPT. It leverages the self-attention mechanism [3] to model natural language and predict future tokens. Another advancement in this area is represented by the Vision Transformer [1], which outperforms CNNs, after being trained on larger amounts of data. The contributions of all of these results cannot be questioned; however, there is an issue of efficiency slowing down Transformers. The self-attention mechanism requires quadratic complexity as the context window grows. Because of this, Transformers require relatively powerful hardware for training and inference, making deployment and usage of such models harder. With little to no change in the primary model architecture, only improvements in benchmarks are gained by increasing parameter count (in the range of billions: Llama 3.1 405B) and using better datasets. Effective models need to be deployed on low-resource hardware such as personal computers, smartphones, and embedded devices.

Recently, there has been an attempt at challenging the Transformer architecture by using State Space Models. These networks have linear complexity, being able to be computed using convolutions or recurrences. Mamba [2] uses Selective State Spaces to selectively propagate or forget information from the context window. It features a hardware-aware design, which enables fast model passes, up to 5 times faster than a Transformer, while also having lower memory usage. There have been studies extending the capabilities of Mamba to image [4] and video [5] tasks. In this project, I'm presenting a Mamba-based model, following the VideoMamba architecture.

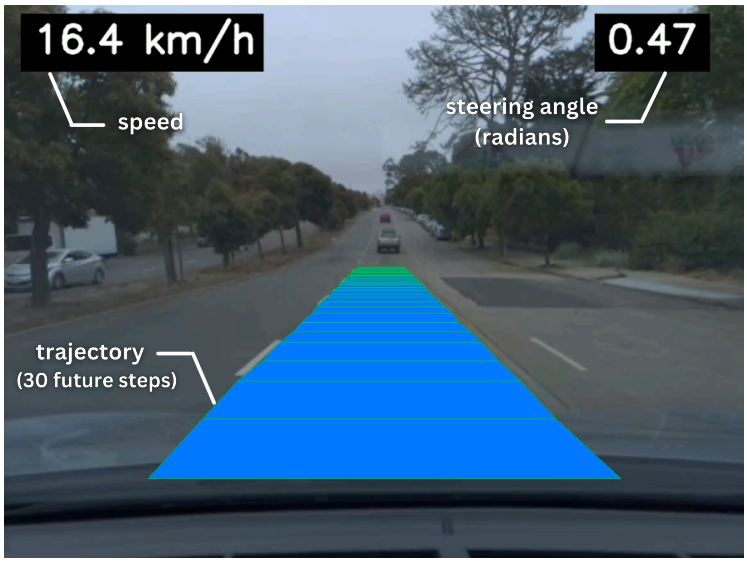

Figure 1.1. An example of the model output at one timeframe.

For a model architecture to be effective, it requires experimental backing, as neural networks need to be observed on real-world data and scenarios. Mamba, being a relatively

new architecture, has little real-world usage, most notably the Mamba 3B language model from the original paper and Codestral-Mamba model from Mistral. I trained my model, **Steer**, on a real-world dataset for self-driving. It processes 10 past frames to predict the future trajectory of the vehicle, along with steering wheel angle and speed. Such a model would be most effective, in a real scenario, deployed on an embedded system inside a car for local inference. For this, a Transformer model would be sub-optimal, making the results even more hopeful.

Steer is an end-to-end neural network with the VideoMamba architecture at its core. In figures 1.1 and 1.2, an example of the model output can be seen. The forward pass takes in a temporal context of the past 10 frames, along with trajectory data, to predict the values. The figure is a visualization generated using a script from the project repository.

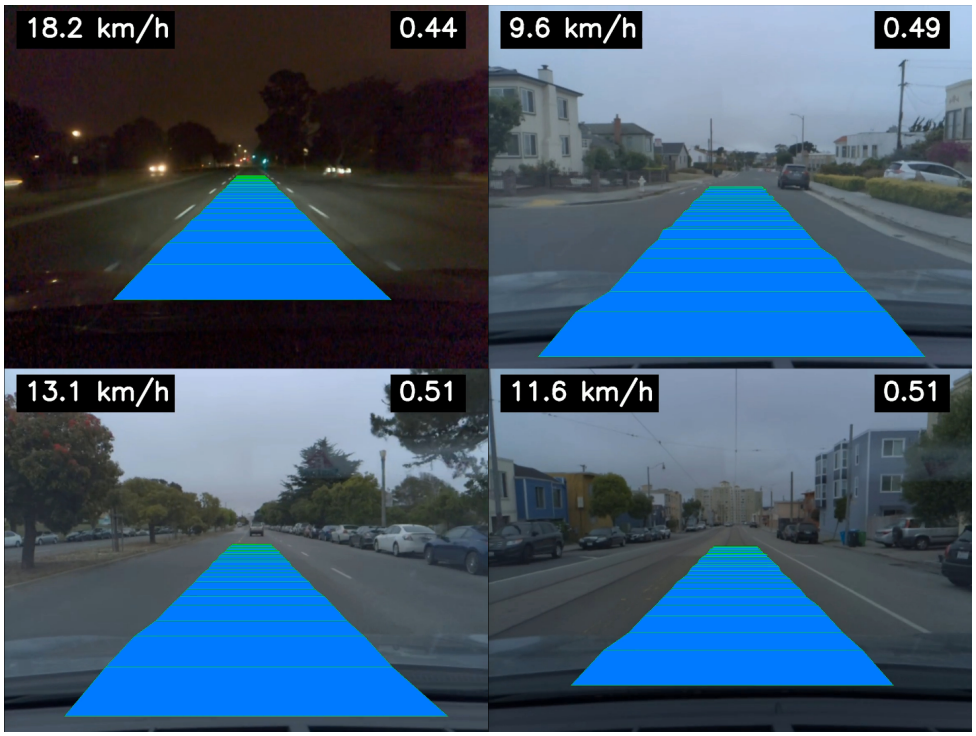

Figure 1.2. Model outputs in different light and enviormental conditions.

This task was intentionally designed to take in multiple inputs and generate multiple outputs, as it challenges the data relationships the model can understand, requiring a deep analysis of the input data. The other two models, used as a reference in benchmarks, proved inferior in predicting all three metrics correctly.

Due to the limitations of our hardware resources, the scope of this project was limited. Not enough training iterations were done to achieve the full potential of these models; however, we consider enough simulations were done to get a correct idea of the capabilities of each model. These findings are explained and expanded upon in the following chapters.

2. Context

2.1 Classical methods

Convolution Neural Networks CNNs [6] are traditionally used for vision tasks, such as image classification and feature extraction. Consider an input feature map $X \in \mathbb{R}^{H \times W}$ and a 2D convolution filter $W \in \mathbb{R}^{K_H \times K_W}$, where K_H and K_W are the kernel's height and width, respectively. The 2D convolution operation with stride S and padding P produces an output feature map $Y \in \mathbb{R}^{H' \times W'}$, where:

$$H' = \left\lfloor \frac{H + 2P - K_H}{S} \right\rfloor + 1 \qquad W' = \left\lfloor \frac{W + 2P - K_W}{S} \right\rfloor + 1 \tag{2.1}$$

Each element of the output $Y[i, j]$, with stride S and padding P is computed as:

$$Y[i, j] = \sum_{m=0}^{K_H - 1} \sum_{n=0}^{K_W - 1} W[m, n] \cdot X[i \cdot S + m - P, j \cdot S + n - P] \tag{2.2}$$

The model's task is to determine the weights of W. The filter slides over the input X spatially. At each position, element-wise multiplication is performed between the filter and the corresponding region in the input, followed by a summation to produce a single scalar output.

CNNs are powerful due to their inherent inductive biases, particularly locality and translation equivalence. Locality ensures that CNNs focus on local patterns, reducing computational complexity and capturing essential features. Translation Equivalence allows CNNs to recognize patterns regardless of their spatial location, ensuring consistent feature detection across the input.

Transformer models Transformer models were introduced by [3], initially for language modeling. It uses the attention mechanism to understand the context and predict the next token. The attention matrix is computed using a "Scaled Dot-Product Attention" operation, with queries, keys and value tensors of dimension d_k:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{2.3}$$

This approach enables greater parallelization, since the attention requires only the global input and not the previous hidden state output, like in the case of RNNs. However, this operation scales quadratically as the context length grows. To generate a sequence, a Transformer requires all previous generated tokens to be appended to the input to generate the next one. This is contrary to an LSTM-like model where the information is memorized and forgotten in a unified hidden state. Many techniques, such as flash attention, have been leveraged to improve the inference speed, however, the scaling laws cannot be overcome.

2.2 State Space Models

SSMs were first introduced in the 1960s in the area of control engineering. Here we are referring to them in the context of deep learning. SSMs define a continuous time-invariant system, with a fixed hidden state $h \in \mathbb{R}^N$, three learnable parameters (A, B, C), and an input $x \in \mathbb{R}^T$ with an output $y \in \mathbb{R}^T$:

$$\begin{aligned} h'(t) &= Ah(t) + Bx(t) \\ y(t) &= Ch(t) \end{aligned} \tag{2.4}$$

$$\begin{aligned} h_t &= \bar{A}h_{t-1} + \bar{B}x_t \\ y_t &= Ch_t \end{aligned} \tag{2.5}$$

To utilize this model, it needs to be discrete, as seen in equations 2.5. The importance of the system being continuous is that it adds a temporal element, as it allows the discrete parameters to be sampled at various intervals. This linear recurrence can be computed efficiently in parallel using a convolution.

The information is compressed into the latent space h. SSM's main benefit is that it scales linearly with context, enabling better performance than Transformers at a larger scale. The theoretical complexity of a Mamba model implemented using FFT is $O(L\log(L))$, while for a Transformer it is $O(L^2 \cdot D)$, where L is the sequence size and D is the embedding dimension.

2.3 Selective State Space Models

Selective State Space Models, also abbreviated as S6, have been introduced by [2] to challenge the Transformer architecture. They add the selection mechanism to SSMs, to better model data relationships. The core idea of this mechanism is to create the matrices as functions dependent of time. This way information can be memorized and forgotten, enabling long context scaling:

$$\begin{aligned} h'(t) &= Ah(t) + B(t)x(t) \\ y(t) &= C(t)h(t) \end{aligned} \tag{2.6}$$

$$\begin{aligned} h_t &= \bar{A}_t h_{t-1} + \bar{B}_t x_t \\ y_t &= C_t h_t \end{aligned} \tag{2.7}$$

SSSMs model Δ_t as a function time, allowing the model to learn how to sample from the continuous system. This adds addresses previous issues, such as the ability to alter the hidden state based on new information. We want to be able to ignore irrelevant information and to reset our hidden state. When $\Delta_t \to 1$, $\bar{A}_t$ approaches 1, thus filtering the token. We can also reset the hidden state, when $\Delta_t \to \infty$, which also makes sense from a time point view, as information is forgotten after much time:

$$\bar{A}_t = \exp\left(\Delta_t A\right) \tag{2.8}$$

$$\bar{B}_t = (\bar{A}_t - 1)\frac{B(t)}{A} \tag{2.9}$$

Mamba, which is an S6 architecture, also benefits from speed improvements, thanks to its hardware-aware design. It leverages a parallel scan algorithm, without expanding the whole state to avoid IO access between different levels of the GPU memory hierarchy (SRAM and HBM).

3. Related Work

End to End Learning Autonomous Driving The first published work to introduce end-to-end neural networks in the field of autonomous driving is PilotNet [7] from Nvidia. They present a model which takes as input a single image frame, from the vehicle's front view, and it computes the steering wheel angle. The core idea is that all the processing is done inside a neural network, a "black box", which does not need to explicitly learn all the features a human might require: lane segments, road surface, etc. It develops all the required logic to successfully map the image pixels to the steering wheel angle. The architecture is split into two main parts: a list of convolutional layers to progressively downsize the input, followed by fully connected layers to eventually output the desired value. This works very well, thanks to the inductive bias of CNNs, translation equivalence, and locality. For the purposes of this benchmark, we modified PilotNet to merge image features, after the convolutional layers, with the past path data, before feeding them into the fully connected layers. In our implementation, the output is a hidden state, discussed further in chapter 6.

Sequence to sequence models In some specific scenarios, having a context of past data can improve future predictions. This is true in the case of self-driving, demonstrated by ReasonNet [8] and other researchers. The main flaw of PilotNet lies in its simplicity, taking as context only the current frame. ReasonNet proposes a model with short-term and long-term memory. This can help the model better understand its environment, for example, the long-term context can help in occlusion scenarios, such as an obstacle being blocked from view by another car. The ReasonNet architecture is quite complex, as it works with more telemetry data: LiDAR, multi-view input, traffic sign data, etc. This can improve the model for a real-world scenario; however, we dismissed these features for our experiment, taking a simpler approach following [9]. Our Sequence to Sequence model has an encoder and a decoder. The encoder extracts the features of each frame, using a RegNet. The decoder uses a GRU network to process a sequence of image features and past paths to compute the final hidden state.

Transformer models The attention mechanism was originally designed to be used in natural language processing tasks [3]. This work was extended by [1] to allow the use of Transformer models in vision and video [10] tasks. It was proven that a Transformer can outperform a CNN in image classification, however, at the cost of a larger dataset and more training. Their ability to condense information such way allowed multimodal models to work, we can have a model that understands both language and vision and can operate on both. ViTs are best used in situations where there are complex and nuanced features, while only presenting marginally better results in common problems. ViTs require more compute resources, since Transformers need to overcome through training their lack of inductive biases. Similarly to Mamba, at the start of training, the Transformer has no positional (or temporal in the case of the Video Transformer) information, since it needs to learn the positional embeddings for the image patches. A similar approach is used by Mamba Vision models [4], however, using a Mamba encoder block at their core instead of a Multi-Head Attention block.

4. Results

To verify our model's capabilities, we implemented two additional models: PilotNet [7] and Seq2Seq [9, 11]. These two models vary in how they harness temporal features and in their complexity. PilotNet uses only the current frame as context, while Seq2Seq uses an encoder-decoder architecture to process past frames. Regarding their architecture, PilotNet uses a series of convolutional layers followed by linear layers, and Seq2Seq uses an encoder in the form of a RegNet feature extractor and a decoder using a GRU network. Our model, Steer, showed promising results in comparison, however we think it could have major improvements with more training and data.

4.1 Benchmarks

We tested the final validation loss, the ability to understand data relationships and computational efficiency.

Figure 4.1. Validation loss values of each model, computed at intervals of 100 iterations.

During the training process, as seen in figure 4.1, the Steer model took significantly longer (6 epochs, 2200 steps) to achieve a major improvement at the loss. This is in contrast to the classic "hockey stick" shape of the loss graphs of the other two models. This phenom is well known for Transformer and Mamba based models [1, 12], since these families of models do not have the inductive biases such as locality. Position information is learned by adding learnable positional and temporal embeddings [10]. As stated in the graph, it takes time for the model to learn these patterns, however, it offers more flexibility in modeling data relationships. Because of this and other details related to the linear nature of the model, it requires a lot more training than a convolution neural network. Providing the model has enough training iterations and data, it can out pass classical models. The final validation loss is comparable to the one of PilotNet, however, as discussed in later sections, with a much better data relationships. The Seq2Seq model had a much higher loss at the end of the training. The final loss values can be found in table 4.1.

Model performance can be observed from table 4.1. Despite the relatively high parameter

Model	Milliseconds/Iteration	Params	MFLOPs	Validation Loss
PilotNet	89,31	830.288	119	29.70
Seq2Seq	363,93	5.940.334	9417	151.80
Steer	177,18	6.409.756	5130	7.16

Table 4.1. Resources of each model used in benchmarks, batch size of 1 and speed measured on an RTX 2060.

count, compared to the other two models, our model achieved good performance with 0.2 seconds per forward pass. Our model showed better computational efficiency than the Seq2Seq model, which also has more parameters. This can be further optimized with unified kernels, however, a couple of optimization techniques were already adopted. First of all, for the core Mamba blocks we used the reference implementation [2], which takes great attention to a hardware-aware design. This implementation is optimized for least GPU bandwidth consumption, reducing the largest bottleneck. Aside from a data oriented design, it also features optimized Triton kernels for efficient scan operation. These Mamba blocks are used in the overall architecture, which is also later optimized by torch using a compilation phase.

4.2 Observations

One of the core challenges imposed by the nature of this task is represented by the computation of multiple output values. The trajectory, steering angle and speed can be seen correlated values: the speed influences the trajectory, the steering angle changes the path, etc. Out of the three presented models, only Steer showed promises in modelling good data relationships. This is exemplified in figure 4.3. The picture presents 3 frames from a recording where a car is stopping at a traffic lighting. In the first frame it has normal speed and trajectory, while in the following frames it is slowing down. The speed has a noticeably lower value, while the steering wheel is only marginally affected. Whereas, with other models, the change in speed would linearly change the steering angle, which is wrong. The trajectory also seems to ”collapse” in the last frame, which is a behaviour completely deduced by the model, as it was not trained on segments with a median velocity of under $10\frac{\text{km}}{\text{h}}$. This behaviour can be considered, however, correct as the model can no longer see a future path at that moment.

Figure 4.2. Model predictions after very early training stages.

Another interesting fact is pointed by the progress made by the model. In the early stages of training, model output can be seen in figure 4.2. First picture shows a completely

random, and wrong, path. Meanwhile, in the second picture, we can already see the model is starting to learn the trapezoid shape of the trajectory. It is expected for the model to take a while to achieve a good default setting. The advantage in our architecture is that by the time the model achieves this small breakthrough (around step 2200), it already figures out decent spatial and temporal embeddings which helps the later improvements. The PilotNet model can be observed stagnating after a few epochs, while this is also due to the constrained training steps and dataset size.

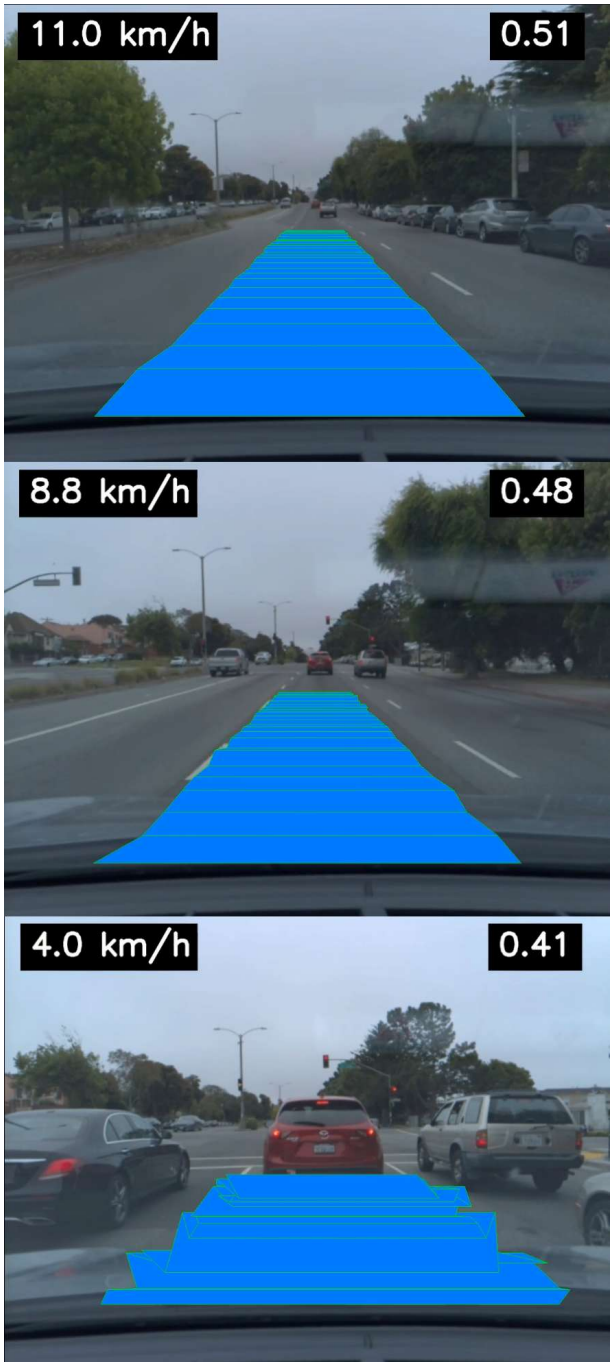

Figure 4.3. Model outputs from selected frames while stopping at a traffic light.

5. Data

5.1 Dataset

The dataset used for training is **Comma 2k19** [13]. It was created by the Comma AI company, and licensed under the MIT licensed. It is arranged in 2019 video segments, each one containing a 1-minute video along with sensor data (relevant here: steering wheel angle and vehicle speed). The data was collected by a car driving along a highway in California, between San Jose and San Francisco. The dataset is about 80GB in total size, and it is split into eight 10GB chunks.

Due to our hardware resource limitations, we chose to work with only the first chunk of the dataset. The model was trained on 4 epochs on only 20% of the first chunk. For the model to actually improve and learn, it requires multiple epochs (full dataset iteration), however our resource didn't allow for us to iterate multiple times over the full dataset. By using only a fragment of the available data, we were enabled to run for multiple "sub-epochs" and get model improvements. As discussed in the Further Work chapter, major improvements in model quality should be achieved if it were trained on more data and for more full-epochs.

5.2 Preprocessing

For efficient training, we needed to prepare our dataset in a ready-to-serve format. This task consisted in organizing the data, choosing useful information, synchronizing sensor data, converting to useful data formats and data cleanup. We wrote a script for doing this work, `prepare.py`, which should be run before the training script. First of all, all the segments from one route are merged together in chronological order, this includes video frames and sensor data. The frames are extracted from videos and saved as compressed images. The sensor data are then synchronized to the frame number. This way, we can extract one model input by using only one timestamp id, which links the current video frame, past frames for context and the current sensor data. The sensor data are written to compressed numpy array files, this way we minimize file size, and we achieve efficient reading and loading. A sample overview of the processed directory can be seen in figure 5.1 below. We are doing a dataset split, 80% training and 20% validation.

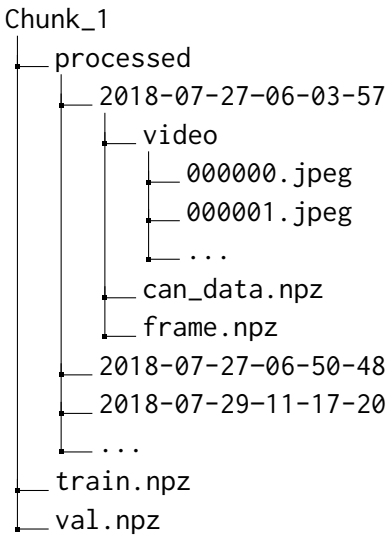

Figure 5.1 An example of a processed Chunk.

5.3 Data cleanup and augmentation

To make the most of the available data, we removed outliers and useless data. In our case, we chose to remove video segments where the car has an average velocity of less than $10\frac{\text{km}}{\text{h}}$,

as well as frames with timestamps that do not allow for the necessary temporal context. For training, we used a context of 10 frames, meaning we could use frames starting only from 11 and going up to the last index minus 30, because we need the next 30 frames to compute the loss. After this, the frames were downsized to 224 by 224. The frame exposure was randomly adjusted for selected frames, in hopes to reduce model biases.

5.4 Visualization

For the purposes of our model, we chose to use only the CAN data as it offered a simple way to synchronize sensor inputs with video frames, and it has a relatively small error margin. The dataset offers post-processed data, as well as GNSS data, as seen in the plot below.

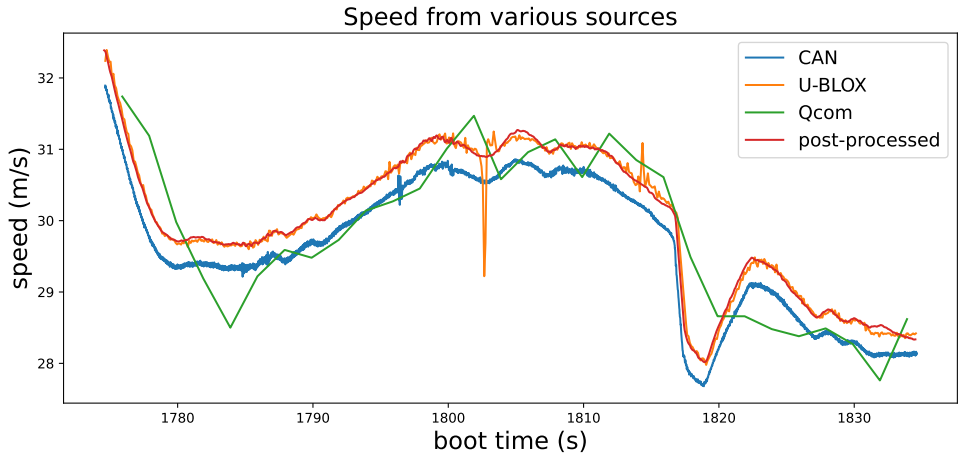

Figure 5.2. Speed data collected from various inputs, during one sample segment.

Here is also a representation of the orientation of the car. The roll and pitch are close to zero, however, the sensors are not perfectly mounted are pruned to errors. The yaw is expected to change as the vehicle is taking a turn.

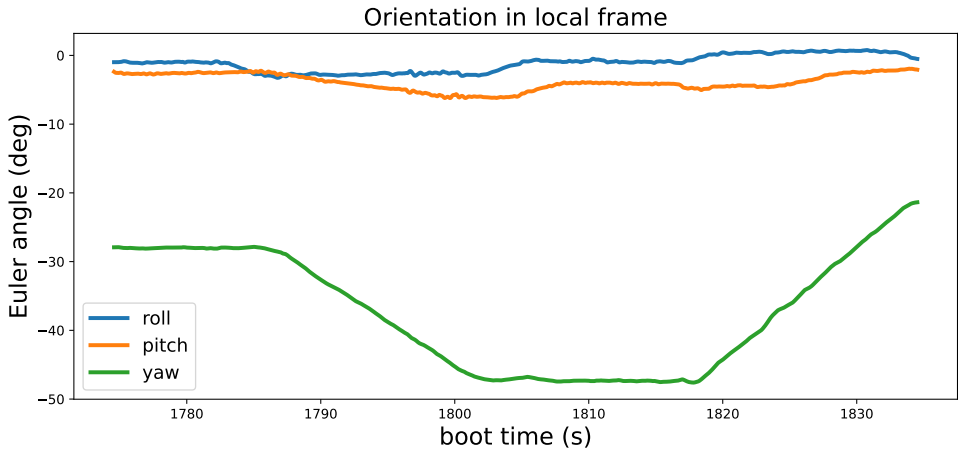

Figure 5.3. Euler angles respective to the local ground plane.

6. Network Architecture

For designing this model, we followed the Video Mamba architecture [5], which is mainly based, on the principles introduced by the ViT model [1].

Input data The model takes as input two tensors. The video context is composed of $T+1$ past frames, the additional one being the current frame, and it is of shape $(B, C, T+1, W, H)$. The past path information is made out of $T+1$ arrays, each with three integers: the x, y and z local-frame coordinates at their specific timestamp.

Output data The model outputs a single tensor of shape $(B, T+1, 3)$, with values for future path trajectory, vehicle speed and steering-wheel angle. The data for the trajectory can be found in the first T elements, each element containing the xyz coordinates. The $T+1$ element has the steering angle at position 0 and the speed at position 1, the last position being left as 0.0. We decided on this approach for the outputs for efficiency, since it is faster to read and move a single numerical tensor, as opposed to a dictionary with three tensors.

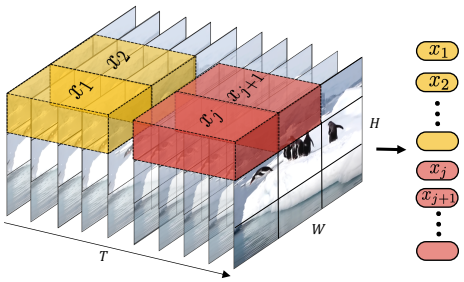

Figure 6.1. Tubelet embedding diagram from the ViViT paper [10].

Encoder blocks For the main Mamba[2] blocks, we used a fork from ViM[4], which added bidirectional scans. This improves pattern recognition in the context of vision tasks. These are implemented using Triton kernels, later also fused with the torch compiler.

Video encoding The first block in the video encoding pipeline is for extracting 3D image patches. This is known as "Tubelet Embeddings", and a visualization can be seen in figure 6. It is achieved by using a 3D convolution of kernel size and stride equal to (K, P, P) (*i.e.*, $1 \times 16 \times 16$). The video $X \in \mathbb{R}^{T \times 3 \times W \times H}$ is resized to $X_p \in \mathbb{R}^{T_p \times C}$, with $T_p = T \cdot \frac{W}{P} \cdot \frac{H}{P}$ and C being the embedding size. Following this, we add two learnable embeddings, one for spatial information ($p_s \in \mathbb{R}^{(\frac{W \cdot H}{P^2}+1) \times C}$: local frame position data) and one for temporal information ($p_t \in \mathbb{R}^{T \times C}$: global video time data): $X = [X_{cls}, X] + p_s + p_t$. After the patches are created, the data is passed through N_1 Mamba blocks, with layer normalization applied before each one and with residual connections.

Path encoding The paths are of shape $(B, T+1, 3)$. As we later need to concatenate the paths and video features, they will need to be of the same shape (B, T_p, C) or $(B, 1, C)$. For performance considerations, since the path information are much smaller than the video, the path features are upscale in two steps. First, we use a linear layer to obtain $P_1 \in \mathbb{R}^{\frac{C}{6}}$, from $P_0 \in \mathbb{R}^{T \times 3}$. This encoding will be passed through N_2 Mamba blocks, similar to the video encoding; however, we are using $N_2 = \frac{N_1}{2}$. After this, the path features are upscale to $P \in \mathbb{R}^{C}$, of shape $(B, 1, C)$. Finally, we add the two features to get the hidden features.

Head For additional processing of the hidden features, we are using an MLP head. The MLP contains two layers with a GELU non-linearity. A dropout layer, of value 0.1, is also used to reduce overfitting. An overview of the whole model can be seen in figure 6.2.

The algorithm is briefly presented in the equations below:

$$\mathbf{X} = [\mathbf{X_{cls}}, \mathbf{X}] + \mathbf{p_s} + \mathbf{p_t}, \qquad \mathbf{X} \in \mathbb{R}^{(T \cdot \frac{W \cdot H}{P^2}+1) \times C}, \mathbf{p_s} \in \mathbb{R}^{(\frac{W \cdot H}{P^2}+1) \times C}, \mathbf{p_t} \in \mathbb{R}^{T \times C} \tag{6.1}$$

$$\mathbf{P} = \text{SSSM}(\text{LN}(\mathbf{P_i})) + \mathbf{P_i}, \qquad \mathbf{P} \in \mathbb{R}^{1 \times C}, \mathbf{i} = 1 \dots N_2 \tag{6.2}$$

$$\mathbf{E} = \mathbf{P_{N_2}} + \mathbf{X}, \qquad \mathbf{E} \in \mathbb{R}^{(T \cdot \frac{W \cdot H}{P^2}+1) \times C} \tag{6.3}$$

$$\hat{\mathbf{y}} = \text{MLP}(E), \qquad \hat{\mathbf{y}} \in \mathbb{R}^{C} \tag{6.4}$$

Computing outputs The output features only represent another hidden state in the model. To compute the three model output values (future trajectory, speed and steering-wheel angle), the hidden state is forwarded to three MLPs. Each one has two linear layers with a GELU non-linearity between them. The layer size of each one is proportional to the output size. This ensures the model can extract the relevant features for each value.

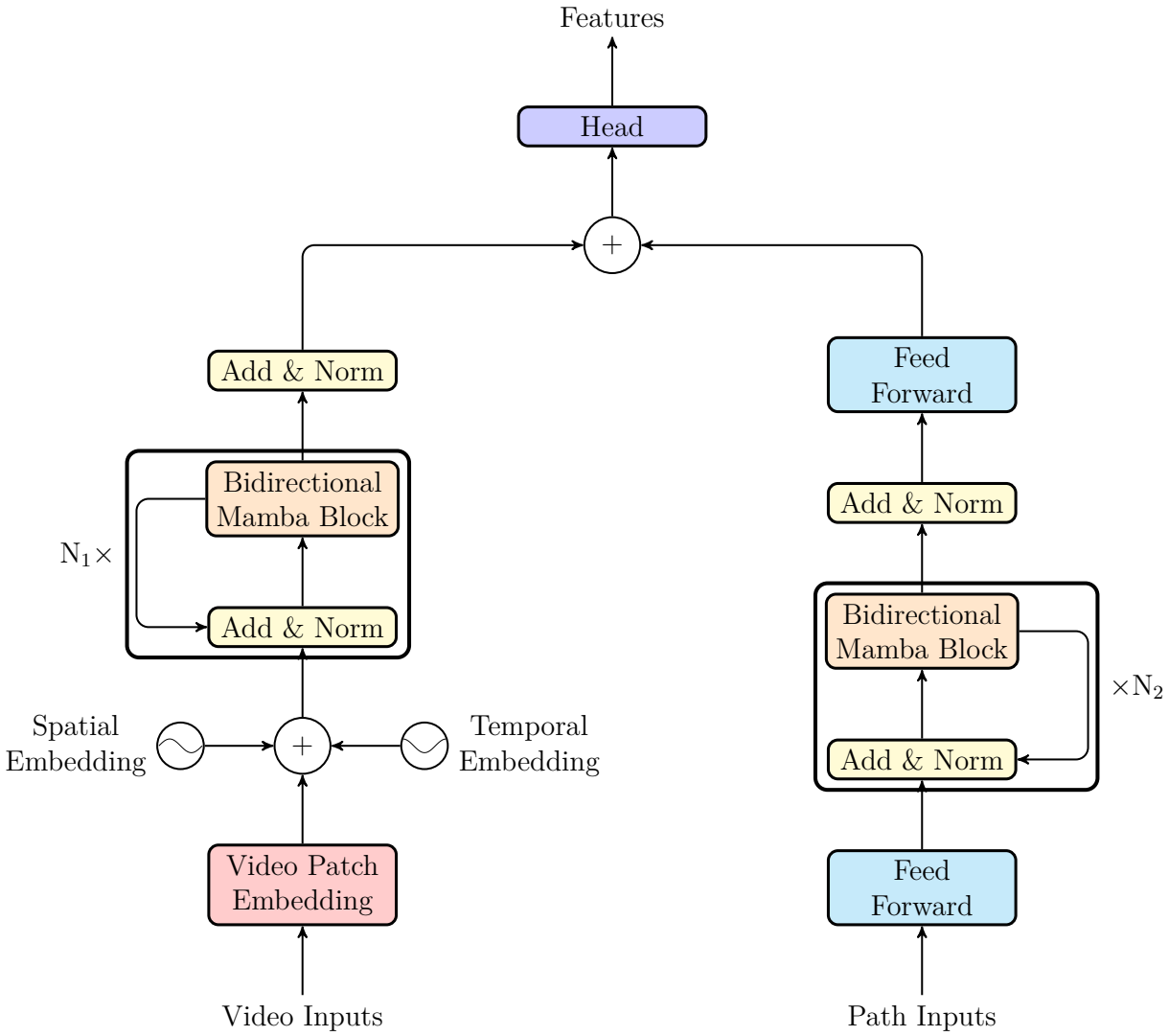

Figure 6.2. Steer model architecture.

7. Training Details

7.1 Optimal Learning Rate

To find the optimal learning rate, we wrote a script that runs a training loop for 500 iterations. The learning rate is linearly changed from $10^{-6.0}$ to $10^{-1.5}$ over the whole process. The loss value was computed and graphed at each step, such that we can analyze what learning rate yields a stable optimization. The resulted plot can be seen in figure 7.1. For readability reasons, the loss value was clipped at 1000. For the analysis of the plot, we are talking in terms of the learning rate exponent. The interval $[-6, -4]$ results in no improvement for the model, meaning the learning rate is too low. The interval $[-4, -2]$ is stable and optimal. The exponents bigger than -2 makes the model explode, leading to no stable optimization.

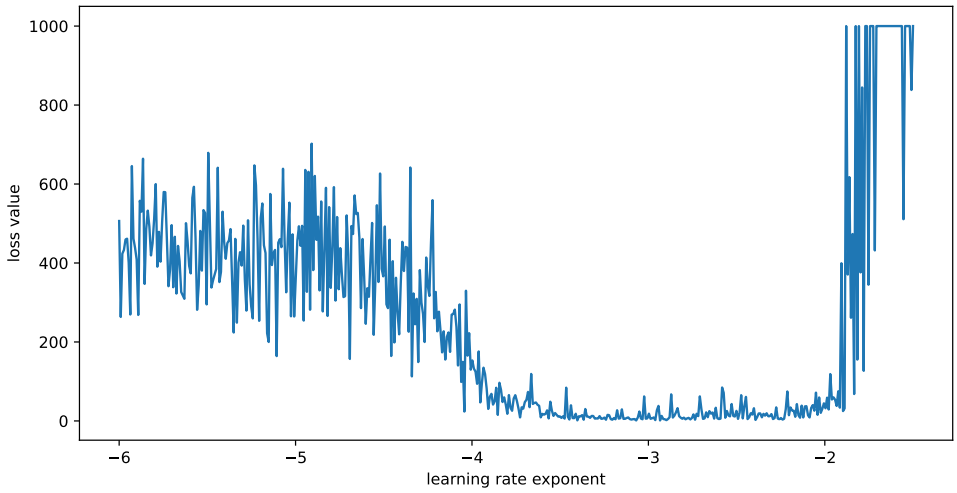

Figure 7.1. Experimental plot for finding optimal learning rate

For the actual training process, we used a minimum learning rate of $5 \cdot 10^{-3}$ and a maximum of 10^{-5}. We did a linear warmum over 3% of the total iterations. After that, we did a cosine learning rate decay, from the maximum to the minimum, over the remaining 97% of the training steps.

7.2 Training process

The model architecture, Mamba, requires more training iterations for good results. Because of this and the hardware limitations, we chose to use only 2% of the available dataset. Considering this, an epoch, with a batch size of 4, takes only 370 steps. We opted for a 10 epochs run, with 3700 iterations. The model was trained on an RTX 2060 6GB GPU. AdamW was used as an optimizer, with $\beta_1 = 0.9$ and $\beta_2 = 0.999$, $\epsilon = \textit{1e-08}$ and a weight decay of 10^{-2}. The results were monitored in realtime using Wandb.

We used an L1 loss for the future path trajectory and MSE loss for steering-wheel angle

and speed. We added these losses, for one final loss value. We chose different loss functions, to emphasize a smaller loss for the trajectory and allow the speed and steering angle to have bigger compared gradients. This is the value we reported, and on which we applied backpropagation.

Regularization Techniques To reduce overfitting, we applied a number of regularization techniques. Dropout layers, with value of 0.1, were used between multiple linear layers. Residual connections were used, between block connections and before and after the encoding sequence. We used learning rate decay, with values as seen in figure 7.2. The gradients were clipped to a value of 5.0, while we also experimented with a value of 1.0, that led to worse results. Layer normalization was applied after each residual connection.

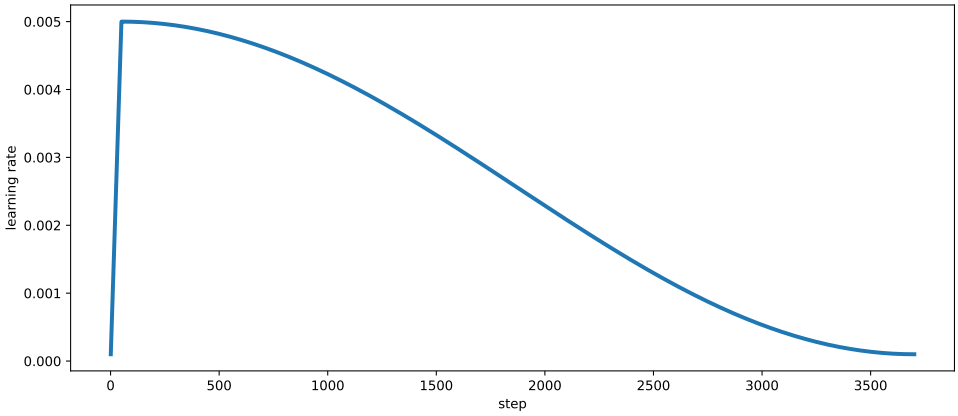

Figure 7.2. The learning rate value over the training process.

Performance Considerations To make the most out of the available hardware, we made a couple of performance optimizations. We used the PyTorch framework, with TF32 and ”high” matrix multiplication precision, instead of ”highest”. The forward pass was cast to bfloat16, and we applied gradient scaling to reduce precision errors. The model returned a single tensor, containing all the values, instead of a dictionary, to improve critical memory operations. During the training process, we measured $\sim 700 \frac{\text{ms}}{\text{iter}}$.

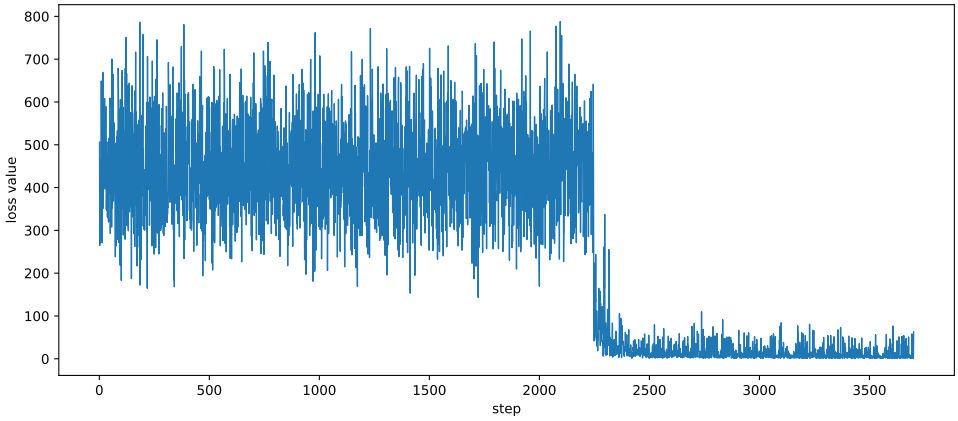

Figure 7.3. Loss values over the training dataset.

8. Conclusion

Our experiment presents a Mamba-based model able to learn how to drive a vehicle. It is efficient compute-wise, with linear complexity and a hardware-aware design. It performs comparable to previous state-of-the-art models, PilotNet and Seq2Seq in our experiments. It has a reduced memory footprint, compared to Transformers and other architectures, allowing it to be used with greater image resolutions and longer temporal context. While it does take longer to train than a foundational model, it proves it is able to achieve better results.

These findings are especially relevant in the context of embedded systems. They require minimal latency and power consumption, such that they require an efficient model for local inference. Our model fits these requirements, with great capabilities. With more work put into selective state space end-to-end models, they could become an universal architecture for on-demand deployment for low-power and embedded devices.

8.1 Further work

For much better results, the model requires more training. It should be trained for at least 30 epochs on the full dataset. It is one of the downsides of the architecture that it requires more training than a classical model, however, the amount of training done was insufficient even for a classical model (CNN). This is due to our limited hardware resources.

Data augmentation should also be used to prepare the model for more diverse scenarios. Besides this, using a driving simulator can much improve the data distribution and training flexibility, as stated in [11]. Mamba-based models scale linearly based on the context, this enables much larger data features, and allows possible token-free models[14]. This is important in a context where we employ a reinforcement learning approach, where the main model learns from a driving simulation, also generated by another model. The Mamba architecture enables the possibility of low-power large models with sufficient context.

Bibliography

[1] Alexey Dosovitskiy et al. “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”. June 3, 2021. DOI: 10.48550/arXiv.2010.11929. arXiv: 2010.11929[cs]. URL: http://arxiv.org/abs/2010.11929 (pages 1, 5, 6, 11).

[2] Albert Gu and Tri Dao. “Mamba: Linear-Time Sequence Modeling with Selective State Spaces”. May 31, 2024. DOI: 10.48550/arXiv.2312.00752. arXiv: 2312.00752[cs]. URL: http://arxiv.org/abs/2312.00752 (pages 1, 4, 7, 11).

[3] Ashish Vaswani et al. “Attention Is All You Need”. Aug. 1, 2023. DOI: 10.48550/arXiv.1706.03762. arXiv: 1706.03762[cs]. URL: http://arxiv.org/abs/1706.03762 (pages 1, 3, 5).

[4] Lianghui Zhu et al. “Vision Mamba: Efficient Visual Representation Learning with Bidirectional State Space Model”. Feb. 10, 2024. DOI: 10.48550/arXiv.2401.09417. arXiv: 2401.09417[cs]. URL: http://arxiv.org/abs/2401.09417 (pages 1, 5, 11).

[5] Kunchang Li et al. “VideoMamba: State Space Model for Efficient Video Understanding”. Mar. 12, 2024. DOI: 10.48550/arXiv.2403.06977. arXiv: 2403.06977[cs]. URL: http://arxiv.org/abs/2403.06977 (pages 1, 11).

[6] Yann LeCun and Yoshua Bengio. “Convolutional networks for images, speech, and time-series”. In: *The handbook of brain theory and neural networks*. MIT Press, 1995, pp. 255–258 (page 3).

[7] Mariusz Bojarski et al. “End to End Learning for Self-Driving Cars”. Apr. 25, 2016. arXiv: 1604.07316[cs]. URL: http://arxiv.org/abs/1604.07316 (pages 5, 6).

[8] Hao Shao et al. “ReasonNet: End-to-End Driving with Temporal and Global Reasoning”. May 17, 2023. DOI: 10.48550/arXiv.2305.10507. arXiv: 2305.10507[cs]. URL: http://arxiv.org/abs/2305.10507 (page 5).

[9] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. “Sequence to Sequence Learning with Neural Networks”. Dec. 14, 2014. arXiv: 1409.3215[cs]. URL: http://arxiv.org/abs/1409.3215 (pages 5, 6).

[10] Anurag Arnab et al. “ViViT: A Video Vision Transformer”. Nov. 1, 2021. DOI: 10.48550/arXiv.2103.15691. arXiv: 2103.15691[cs]. URL: http://arxiv.org/abs/2103.15691 (pages 5, 6, 11).

[11] Eder Santana and George Hotz. “Learning a Driving Simulator”. Aug. 3, 2016. arXiv: 1608.01230[cs,stat]. URL: http://arxiv.org/abs/1608.01230 (pages 6, 15).

[12] Rui Xu et al. “Visual Mamba: A Survey and New Outlooks”. July 6, 2024. arXiv: 2404.18861[cs]. URL: http://arxiv.org/abs/2404.18861 (page 6).

[13] Harald Schafer et al. “A Commute in Data: The comma2k19 Dataset”. Dec. 13, 2018. arXiv: 1812.05752[cs]. URL: http://arxiv.org/abs/1812.05752 (page 9).

[14] Junxiong Wang et al. “MambaByte: Token-free Selective State Space Model”. Aug. 9, 2024. DOI: 10.48550/arXiv.2401.13660. arXiv: 2401.13660[cs]. URL: http://arxiv.org/abs/2401.13660 (page 15).