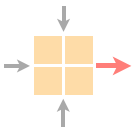
ETH zürich



Networked Systems
ETH Zürich — seit 2015

Advancing Predictions for Video Streaming with Transformers

Semester Thesis

Lukas Röllin

Tutor: Alexander Dietmüller, Jacob Romain

Supervisor: Prof. Dr. Laurent Vanbever

January 2022

Abstract

The Transformer is a sequence-to-sequence (seq2seq) neural network architecture that has proven itself useful for a wide variety of applications. We compare the Transformers performance to several baseline models on a video streaming task to predict transmission times. At the moment the models used for this task are not optimized for seq2seq predictions. To address this we use the Transformer which is tailored to efficiently find long-term dependencies in the data.

We find that the Transformer does not significantly outperform the other models. We suspect a lack of long-term dependencies in our dataset or the lack of essential features to find those dependencies. Nevertheless the Transformer shows better performance than the other models for the tail loss. A transformer variant using probabilistic regression is able to marginally outperform the other models in the mean loss. Additionally we describe the adaptations we made to the Transformer to make it compatible with multi horizon timeseries prediction tasks.

Contents

1 Introduction **1**

2 Background and Related Work **3**
2.1 Background . . . 3
2.1.1 Transformers . . . 3
2.1.2 Video Streaming . . . 4
2.2 Related Work . . . 5

3 Design **7**
3.1 Problem Statement . . . 7
3.2 Puffer Model . . . 8
3.3 Probabilistic Regression . . . 9
3.4 Predict 5 Future Steps . . . 10
3.5 Transformer . . . 10
3.6 Probabilistic Transformer . . . 11
3.7 Loss functions . . . 12

4 Evaluation **14**
4.1 Comparison of the Original Task . . . 14
4.2 Prediction Window Problem . . . 15
4.3 Larger Prediction Windows . . . 16
4.4 Training Data Size . . . 17
4.5 History Window . . . 18
4.6 Influence of the Pretrain Dataset Size . . . 18
4.7 Start Up vs Running Flows . . . 20

5 Outlook **23**

6 Summary **24**

References **25**

A Multi-GPU Setup **I**

Chapter 1

Introduction

Motivation Machine learning has been successfully applied to many networking domains including video streaming. Machine learning models can be used for a variety of video streaming tasks. The one we focus on in this thesis is the prediction of transmission times for video chunks. Those predictions are important for video streaming algorithms as they need to know how long it takes for a chunk to be loaded so they can make good decisions on which chunk quality to request next. For video streaming it is important that the performance is not only good on average, but also at the tail. One bad prediction can cause the algorithm to request chunks that are too big and cause the stream to stop and wait for the chunks to be downloaded.

There is a paper from Stanford University [15] where they built a video streaming platform to test their video streaming algorithms. In an ongoing experiment, they collect extensive networking data from those video streaming session and generously make them available online. Especially nice is that this dataset is not synthetic and contains data from actual users, which means that it is much more representative of the real world performance. This makes it the ideal candidate for us to test our machine learning models. We expect there to be long term patterns in the networking data that announce changing networking conditions before they happen. We expect this as other machine learning models used in the Puffer project already are able to predict future network conditions quite accurately. What we want to improve is using a machine learning architecture that is more specialized for such patterns.

There is a class of models called sequence-to-sequence (seq2seq) models. As the name suggests they are used for problems where we have an input sequence that has to be mapped to an output sequence. One example would be natural language translation tasks where we have a sentence (input sequence of words) that we would like to map to the translated sentence (output sequence of words). The Puffer dataset consists of video chunks that each contains information about the networking state and the time it took to send them. For our application we have the information about the previous few chunks available and we need to predict the transmission times for the next few chunks in the series. This means we have a input sequence with the information from the previous chunks that we would like to map to a output sequence of future transmission times. Because of this similarity we decided to use a seq2seq model.

As of 2021, the Transformer is a promising new seq2seq neural network architecture with models like GPT-3 [7] being successfully applied to a variety of different tasks. Transformers were first used for natural language processing, especially for translation tasks. Over time they were found

to be useful for a wide variety of problems and there are whole websites showing applications with GPT-3 [1]. The Transformers ability to detect long-term patterns (dependencies between elements in the sequence that are far apart) is the reason we use it for the transmission time predictions.

Task and Goals The goal of this thesis is to find out whether Transformers are well suited for network transmission time predictions. We want to investigate how the Transformers need to be adapted to work for multi-horizon time series prediction tasks. We also want to gain insights into the Puffer dataset and its implications on the use of seq2seq models for video streaming tasks.

Overview In Section 2 we describe the Transformer architecture as well as some video streaming basics. Section 3 describes the work that was done to get the models running and what we did to optimize the performance. In Section 4 we compare the Transformers performance to the original model as well as a probabilistic regression model. In Section 5 we go into other ideas that we think would be interesting to investigate. In Section 6 we give a short summary of the thesis.

Chapter 2

Background and Related Work

2.1 Background

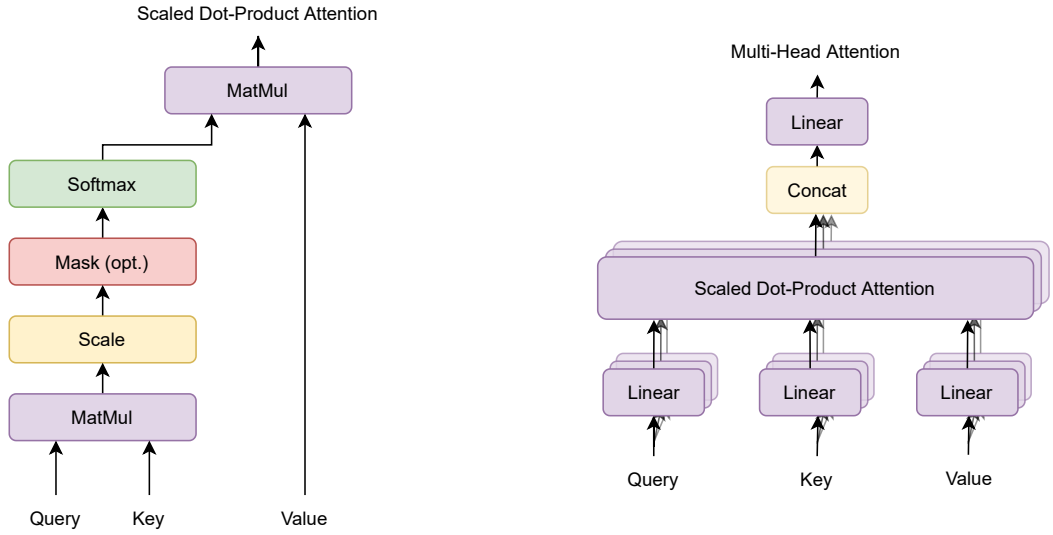
2.1.1 Transformers

To understand this thesis we first give a short overview of the Transformer architecture [12].

Attention To efficiently learn dependencies between elements in the sequence, the Transformer exclusively relies on a mechanism known as 'attention'. The attention mechanism consists of a query, key and value vector for each sequence element. Those are generated by multiplying the embedding vector of each sequence element with three learned matrices. In Figure 2.1a the function applied on the left hand side is called the compatibility function between the query and the key. For each element in the sequence we calculate the compatibility function between its query vector and all key vectors. We call this output the compatibility vector which tries to capture how much attention the query element should pay to each key element. We then apply the dot product between the compatibility vector and the corresponding value vectors. For each sequence element we get an attention vector out, that is a linear combination of all value vectors weighted by the compatibility vector. Because each element has both its own key and query vector, the attention is asymmetrical. That means the attention between two elements changes depending on which of the elements acts as the query and which as the key. Because the Transformer uses only this attention mechanism, it can be trained much more efficiently than models with recurrent neural network (RNN) layers, in particular if there are long-range dependencies in the data. This is the case as RNN layers need to be trained in sequence while attention layers can be trained more parallel.

Multi-Head Attention For the Transformer, multi-head attention layers are used which consist of multiple attention layers running in parallel with multiple independent sets of learned matrices. This multi-head attention is used so that the model can jointly learn different types of attention. The structure of the multi-head attention layer can be seen in Figure 2.1b.

Positional Encoding Because the attention mechanism processes all elements in parallel, any information about the order of elements is lost. Positional encoding is applied so that the Transformer can make sense of the order of the sequence. It is used to inject information about the position of an element. The positional encoding used in the original paper is made up of sine and cosine functions with different frequencies that are added to the sequence. Because each position gets a different frequency the model should be able to differentiate them.



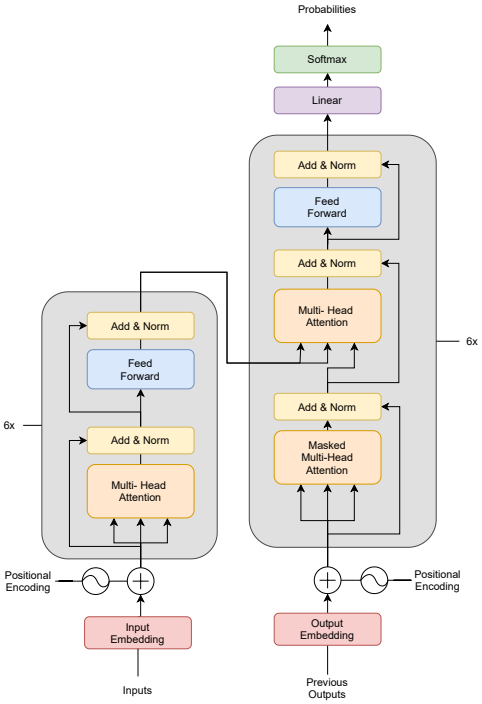
(a) Scaled Dot-Product Attention (b) Multi-Head Attention

Figure 2.1: Attention Diagrams inspired by the Transformer Paper [12]

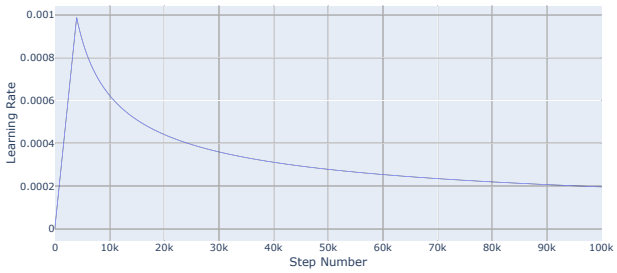
Model Structure We go into the general elements of the Transformer architecture and give an overview of the structure in Figure 2.2a. The Transformer can be separated into two parts, an encoder stack and a decoder stack. On the encoder side we apply the sequence of inputs and transform it into an embedding that the Transformer can use. Additionally we add a positional encoding to the sequence so that the Transformer can deduce the order of the sequence. This then gets into the encoder layer that is made up of the attention layer, batch normalization and feed forward layer. We can put multiple of those encoder layers in sequence. The original Transformer uses 6 of those layers. For the decoder side we feed in the previous predicted outputs which are also transformed to an output embedding. The output also gets a positional encoding added to it. Afterwards we have decoder layers that are similar to the encoder with the difference that the output from the encoder side also gets fed into the second attention layer. This is done so that each position in the decoder can also attend to every element in the input. We can again change the number of decoder layers we put in sequence. At the output there is a linear layer and a softmax layer to predict the next output. This can be changed to get the output into the needed format. Furthermore in the paper they use a customized adam optimizer with a variable learning rate. The learning rate increases linearly until it reaches the number of warmup steps and then decreases to the square root of the step number. The learning rate can be seen in Figure 2.2b for a model dimension of 256 and warmup steps of 4000. A dropout rate of 10% is applied to the output of each sub layer.

2.1.2 Video Streaming

We are interested in using the Transformer for the prediction of transmission times in video streaming. Videos are not streamed as one big block but they are cut into small pieces called chunks. This chunking allows it to change the stream quality on the fly depending on the network conditions.



(a) Diagram of the Transformer inspired by the Transformer Paper [12]



(b) Learning Rate visualized over 100k Steps

Figure 2.2: Diagrams to illustrate the Transformer

This is done by having multiple qualities available for each chunk and the client deciding which quality should be loaded next. To get a continuous stream without interruptions, there is a buffer that can store the next few video chunks such that we can already load the next parts of the video. This is done such that we do not need to pause the video playback and wait for the next chunk to arrive if it should arrive too late. The difficult part for the clients streaming algorithm is to pick the best video quality chunks that can be transmitted such that the buffer never gets empty and the video never has to be paused. For this we need a good estimate on how long it takes for the next requested chunk to be transmitted. Previously, linear regression and fully connected neural network models were used. We would like to use the Transformer instead.

2.2 Related Work

This paper uses the data generated by the Puffer project [15]. It is also used as a starting point for comparisons between the Transformer and the model that was used in their project. Several other papers have used the Transformer for time series forecasting. In [14] the Transformer is compared to other time series prediction models. The models are tested on predicting the influenza prevalence for four future time steps using the last 10 as the input sequence. The Transformer outperforms the other tested models. We found this paper to be similar in structure to ours as they are also adapting the Transformer for time series prediction but just for a different task. Another paper using Transformers for time series forecasting is [10]. In the paper they found out that using a

probabilistic output instead of point estimates can improve the training and performance of the Transformer. They also found out that changing the layer sizes and number of encoder/decoder layers did not really affect the performance of the Transformer. We found both of those insights to be also true in our case. [8] improves on the attention layer of the Transformer to make the Transformer less affected by anomalies and get faster training times. They use a convolutional attention layer for the first problem and restrict the attention connections for the second problem. While we did not use those methods in our work, it helped us understand that the Transformer might be more susceptible to anomalies.

Chapter 3

Design

Our objective is to compare Transformers with traditional models, using the Puffer [15] dataset as a benchmark. We use a recreated Puffer model as well as several custom models: a probabilistic regression model, a Transformer and a model that is a combination of the two.

Frameworks The models described in the following parts were built with the help of PyTorch [2], PyTorch Lightning [3], scikit-learn [4] and scikit-optimize [5].

Overview In Section 3.1 we go into the general problem structure while we describe the process of recreating the Puffer model in Section 3.2. In Section 3.3 we describe the probabilistic regression model which came as an idea while working on the recreated model. In Section 3.4 we explain the changes for predicting 5 timesteps. In Section 3.5 we describe our work on getting the Transformer running for our task. Additionally in Section 3.6 we describe a small adaptation of the Transformer model by combining it with parts of the probabilistic regression model. Finally in Section 3.7 we go into the loss functions used to compare the models.

3.1 Problem Statement

Task The task of the model is to predict the transmission times of future chunks by using the information from the previous chunks and the size of the requested chunk. The size of the requested chunk is necessary as we use the transmission time which is highly dependent on the chunk size. The transmission time estimate is then used by the streaming algorithm to make decisions on the requested chunk qualities. This thesis only focuses on the transmission time prediction part.

Dataset The dataset is a collection of video streams consisting of sent video chunks that we need to transform into a usable format for our models to make them work. Each video chunk consists of the transmission time and chunk size as well as some networking information. The networking information is a subset of the tcp_info struct from the linux kernel containing the delivery rate, congestion window, packets in flight, minimum round trip time and the round trip time. We use those data points and convert them so that we have one sample per chunk. Each sample contains all mentioned features for a fixed amount of previous chunks, we call the history window. Likewise each sample has a prediction window which contains the chunk size and the transmission times for a fixed amount of future chunks. In the Puffer paper the history window has a size of 8 and the prediction window has a size of 5.

3.2 Puffer Model

We start by recreating the Puffer model, a simple fully connected neural network model using a discretized probabilistic output.

Structure The model used in Puffer [15] is a neural network with 3 linear layers and RELU units in between. The general structure can be seen in Figure 3.1a. For the output we do not just have the transmission time but a discretized probability distribution. This is achieved by splitting the expected output space into small bins and estimate the probability for a transmission time to be in that bin. Concretely bins of size 0.5s are used with the first bin starting from 0s only being half the size and the last bin going to infinity. A total of 21 bins are used and can be seen in Table 3.1. We use the same model structure in this paper but we optimized some of the hyperparameters as seen in the hyperparameter tuning paragraph below.

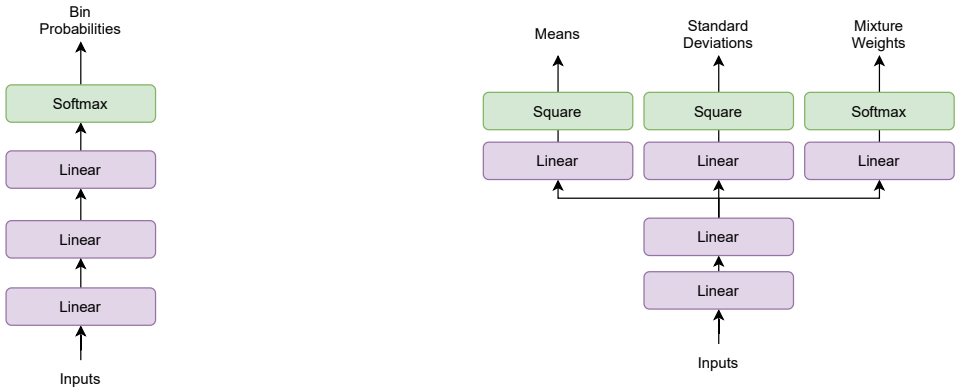
Bin 1	Bin 2	Bin 3	Bin 4	...	Bin 20	Bin 21
$[0s, 0.25s)$	$[0.25s, 0.75s)$	$[0.75s, 1.25s)$	$[1.25s, 1.75s)$	...	$[9.25s, 9.75s)$	$[9.75s, \infty)$

Table 3.1: Bins of the Puffer Model

Data Preparation Before we can use the model we need to prepare the data for training and validation. We start by taking the features for the last 8 timesteps and flatten them. Then we concatenate the information of the chunk to be predicted to those 8 timesteps but we leave out the transmission time, so we have a input vector of size 62. If not all timesteps are available we just pad the missing ones by duplicating the oldest available one to the ones missing.

Prediction To get from the bins and its probabilities to the actual transmission times there are two possible approaches. In this paper we refer to them as point estimate and probabilistic approach. The point estimate approach is to take the middle of the most likely bin as the prediction time. The probabilistic approach is to calculate a weighted mean over the bins with the weights being the probability of each bin and the values being the middle of the bins. So as an example we take 3 bins with the bin size beeing 2s. Given the bins $[0s, 1s)$, $[1s, 3s)$ and $[3s, \infty)$ and probabilities 0.2, 0.2 and 0.6 respectively we would calculate a value of $3s + 2s/2 = 4s$ for the point estimate while we get $0.5s * 0.2 + 2s * 0.2 + 4s * 0.6 = 2.9s$ for the probabilistic approach. For the last bin we just add half the bin size to get the "middle" as the intervall to infinity does not really have a middle. For our testing we use the probabilistic approach as it clearly outperformed the point estimate.

Hyperparameter Tuning To find the optimal hyperparameters we optimized over the bin size, number of bins, layer size and learning rate. The number of bins in the original paper were kept rather small to speed up the calculation of the model predictive control. We do not have that limitation so we ran a bayesian hyperparameter search algorithm from scikit-optimize [6] to find the optimal parameters. The best configuration we could find was a binsize of 0.078s, 100 bins, layersize of 267 and learning rate of 10^{-3}.



(a) Recreated Model (b) Probabilistic Regression Model

Figure 3.1: Architecture Diagrams inspired by the Transformer Paper [12]

3.3 Probabilistic Regression

We also use a probabilistic regression model that predicts a gaussian mixture distribution for each input. This can be seen as the continuous equivalent to the bin output of the recreated model.

Structure The probabilistic regression model provides a continuous probabilistic density function in the form of a gaussian mixture distribution as the output. A gaussian mixture distribution consists of multiple gaussian distributions that are weighted and added together to create a new distribution.

$$f_X(x) = \sum_{i=1}^{n} \alpha_i * \frac{1}{\sqrt{2\pi\sigma_i^2}} * \exp\left(-\frac{(x-\mu_i)^2}{2\sigma_i^2}\right) \qquad \sum_{i=1}^{n} \alpha_i = 1 \tag{3.1}$$

The idea is that with enough of those mixture components we can emulate nearly every distribution we need. We can think of it as having a "unlimited" number of bins for our normal model. We changed the output layer of the recreated model to have 3 seperate linear layers that output the means, standard deviations and mixture weights of the gaussian mixture components. For our testing we use 10 mixture components and the negative log likelihood loss for training. The general structure can be seen in Figure 3.1b.

Data Preparation We can use the exact same data preparation as seen above in Section 3.2 for the Puffer Model.

Prediction For the standard deviations we square the output to only get positive values and for the mixture components we add a softmax layer to get the sum of the weights to 1. This is needed because it would not be a probabilistic distribution function without those two adjustments. We also square the means to get only results bigger than 0 as transmission times should also be bigger than 0. But we found that it does not make any difference for the performance if squared or not

as the model will learn this fact out of the data itself. We left the squaring in so we do not have to deal with invalid outputs smaller than 0. The resulting probability density function still goes from minus infinity to plus infinity, but we calculate our predictions as the weighted sum of the means with the mixture components as weights. Since both of them are positive, the prediction will also be positive.

3.4 Predict 5 Future Steps

We want the model to predict 5 timesteps into the future as done in Puffer. For the recreated model and the probabilistic regression model this means replicating the model 5 times and training every model on a specific future timestep. The only thing that we need to adapt in the input of the models is to change the chunk size to the one we want to predict with this model.

3.5 Transformer

Even though the Transformer is useful for sequence to sequence predictions we still need to change multiple things to get it working with our data.

Structure We use mostly the same structure as the original Transformer, but we add linear layers that transform our sequence elements to higher dimensional embedding vectors. A diagram of the model can be found in Figure 3.2a. For the prediction we have all features available on the encoder side but not on the decoder side. This is because our model should not have access to the future networking features as they are not available at the time of prediction. We could let the model predict the future networking features such that they are available on the decoder side. We do not want to do this as the model then would need to predict things we are not directly interested in. Together with the need of having the same embedding dimensions on both the encoder and decoder side, we use two different linear layers for them. The decoder side linear layer transforms the output vector with less features to the same dimension as the encoder linear layer does with the input vector that has all features. For our model the embedding dimension is 256.

Data Preparation There are two possible ways to create the sequences from the dataset: using each feature separately as an element of the sequence or using each chunk with all its features as one element. The first option is to give the Transformer all features described in Section 3.1 separately but we found that this did not perform great. Also this increased the training time significantly as the input size grew by a factor of 7. The method we use considers each chunk to be one element in the sequence. We could then just use the features of this chunk as the embedding but we think the Transformer needs a much higher dimensional embedding vector (the original Transformer used an embedding dimension of 512). The added linear layers described above accomplish this task.

Prediction In the same way we did with the other models, we tested different output formats. We tested the transmission time as the direct model output and the binned approach seen for the recreated model. With the transmission time as the direct model output we saw the same thing as with the Puffer model that it did not result in good performance. For that reason we use the binned approach and set the bins to be equal to the recreated model being 100 bins with size 0.078s. Trying to optimize the size of the bins there were only small improvements that came probably down to run-to-run variance so we used the same value for both models to better compare the

results. We again use the same probabilistic approach as with the Puffer model to get from the bins to the actual transmission times we want to predict.

Decoder Input Another problem we found was feeding back the previous outputs to the decoder. When feeding back all the previously predicted outputs we need to use the target mask to mask the elements we did not predict yet as the Transformer still needs the same dimension for all decoder inputs. We found the performance to be bad for the fourth and fifth timesteps when using this approach. We think this might be down to the Transformer getting confused by its previous outputs and adding up all the previous errors to make the next prediction. We use the previous output now in such a way that we just feed back the most recent prediction. This allows us to get better performance for the fourth and fifth timesteps while still getting the same performance for the other timesteps. Additionally to the last prediction, we also give the model the size of the chunk it needs to predict for the same reasons already explained in Section 3.1. We use the same sine cosine positional encoding described in Subsection 2.1.1 but only for the input.

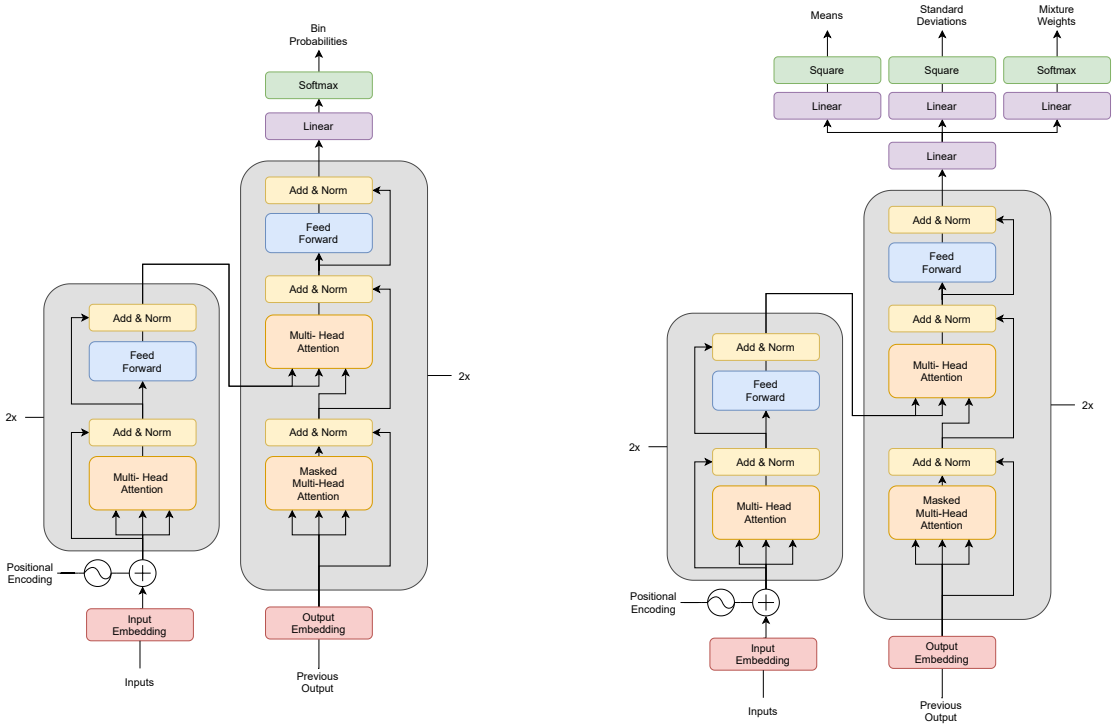
Training For the input we use the source mask to force the Transformer to use the information in the order they appear in. The source mask is specific to the pytorch implementation and masks future positions of the sequence by setting them to minus infinity. It accomplishes that the attention mechanism for each element can only attend to elements that appear before it in the sequence. We also ran it without the mask and there was no measurable difference in performance. The same thing can be said about the optimizer. We first used the standard pytorch adam optimizer with a constant learning rate. After some literature research we changed it to the same optimizer that is used in the original Transformer paper. It is a adam optimizer with customized parameters and the variable learning rate shown in Figure 2.2b. The performance differences are minimal at best. We use it mainly because it is used in the original Transformer. Another thing we had to decide for training the Transformer is if we want to give the decoder the transmission times it predicted or the ground truth. Giving the model the ground truth during training is called teacher forcing [13]. We use teacher forcing for the first epoch, then for the second epoch we use it for 50% of the samples and then afterwards we use only the previous predictions. The model has first one epoch to train on the correct data and then one to change from ground truth to its own predictions. We also let it run with a 50% chance in all epochs and the performance was the same.

GPU Training To be able to train larger data sizes we wanted to also make the model capable of training on multi-GPU setups. We ran into problems with the different parallelization strategies provided by pytorch. We describe the problems in more detail in Appendix A.

3.6 Probabilistic Transformer

Analogous to the recreated model, we try out how the Transformer performs if we use a continuous probabilistic output instead of the binned outputs.

Structure We call this model the probabilistic Transformer and we provide an overview of it in Figure 3.2b. It is built exactly the same as our other Transformer except we changed the last layer to multiple linear layers that predict again the means, standard deviation and mixture components of a gaussian mixture probability distribution function. They are squared and scaled exactly the same as in the probabilistic regression model seen in Section 3.3.



(a) Transformer (b) Probabilistic Transformer

Figure 3.2: Architecture Diagrams inspired by the Transformer Paper [12]

Data Preparation The same data preparation as seen in Section 3.5 for the standard Transformer is also used for the probabilistic Transformer.

Prediction The transmission time gets calculated by multiplying the means of the gaussian mixtures with the mixture components. We use 10 mixture components as we already used for the probabilistic regression model.

Training We found that we could not get the model to converge when trying to train all layers at the same time. So instead we used our trained Transformer and just swapped out the last layers. We freeze the Transformer layers and train only the final linear layers. With this approach we get a model that converges similarly to the other models. We use the negative log likelihood loss like we used for the probabilistic regression model.

3.7 Loss functions

As validation loss functions we used crossentropy, mean square error, accuracy, 99th percentile crossentropy and square loss, but found that only the mean square error and 99th percentile square loss are appropriate loss functions as detailed below.

Training For the training step we use the crossentropy loss where each bin is considered to be a category that we would like to predict. So the loss gets bigger the lower the predicted probability of the correct bin is. For the probabilistic regression models we use the negative log likelihood loss. We can think of it like having the crossentropy loss with an "unlimited" number of bins.

Validation For the validation step we used 6 metrics, but found out that some of them are not really useful. The 6 metrics are accuracy to predict the correct bin, crossentropy loss, 99th percentile crossentropy loss, probabilistic MSE loss, point estimate MSE loss and probabilistic 99th percentile square loss. We found the accuracy to predict the correct bin to be a bad metric as it highly depends on the size of the bins. If the bins are smaller, then it is much harder to predict the correct bin. The same can be said about the crossentropy loss as well as the 99th percentile crossentropy loss. The crossentropy loss is still usefull for the training step, it is only bad if we try to compare the performance of models with different bin sizes. The point estimate MSE loss is taken by using the point estimate approach and calculating the MSE between the prediction and the ground truth. There is nothing inherently wrong with this loss but the performance is worse for all cases than using the weighted average approach (probabilistic approach) to calculate the transmission time. Since both losses compute the MSE against the ground truth, we can just decide to use the better approach as we can decide ourself how we calculate the actual predicted transmission time. The 99th percentile square loss is also based on the weighted average transmission time calculation. We use here the 99th percentile instead of the maximum as it is less susceptible to outliers. We use this metric because we are also interested in how the models perform in the tail case. In networking applications like video streaming the tail performance is important as we not only care how good the experience is for most consumers but we also want to create a good experience for all consumers.

Chapter 4

Evaluation

We first compare the results of our models for the original task of predicting 5 timesteps in Section 4.1. We go into the problem of predicting flows that end inside the prediction window in 4.2 and evaluate the models on bigger prediction windows in 4.3. We take a look into the effects of the training dataset size in 4.4 and investigate the influence of the history window size in 4.5. In Section 4.6 we discuss the influence of pretraining for the probabilistic Transformer. Finally we show the differences between predicting transmission times at the start of a flow and during normal operation in Section 4.7.

4.1 Comparison of the Original Task

We find when comparing the Transformers to the other models that the Transformers slightly outperform the other models in the tail performance and have a similar mean performance.

Data Preparation We start by comparing the general performance between the models on the same problem task as seen in the Puffer paper. This means 800k of samples for the training set and 200k samples for the validation. The sets are taken from the complete dataset collected on the 27.07.21. Those are around 4.3 Million samples from which we use the train test split function from sci-kit learn to split the data randomly into the two sets of the stated size. The split function takes a seed so that we can guarantee comparable results between multiple runs. Additionally for the evaluation we remove all the samples that do not have at least the prediction window available. So in the case of a prediction window of 5 we discard all the samples that do not have at least 5 chunks left until the end of the stream. The reason to do this will be discussed in Section 4.2.

Convergence For the training of all the models we use early stopping. The recreated model and the probabilistic regression model are evaluated every 10 epochs and have a early stopping patience of 3 while the Transformer gets evaluated every epoch and has a early stopping patience of 5. The Transformers get evaluated every epoch due to the much longer training times needed per epoch. To not stop the training process too fast we increased the patience to 5. The Transformers normally stop at around 25-30 epochs. We searched in other papers how many epochs other Transformer implementations take and found a paper on training Transformers [11] that uses bigger datasets where the Transformer converges in only 10 epochs. So we are confident that our early stopping routine should give the model enough time to reach convergence.

Results The overall results can be found in Table 4.1. We see that the recreated model is the worst of the four models as it gets consistently worse results for nearly all experiments we ran. The probabilistic regression model and the probabilistic Transformer are pretty similar in the mean performance. The Transformers have a small lead in the tail performance metric. This probably comes from the fact that the Transformers might be more suited to find long term patterns in special samples that differ from the majority of samples.

Model	MSE	Tail Loss
Recreated Model	0.0992	1.0631
Probabilistic Regression	0.0932	1.0371
Transformer	0.0956	**0.9711**
Probabilistic Transformer	**0.0925**	0.9915

Table 4.1: Performance for the same Scenario as in the Puffer Paper

4.2 Prediction Window Problem

During evaluation we found that the prediction performance was extremely different if we used all samples which had at least the complete prediction window available in comparison to having also samples that only have a subset of the prediction window available as the ground truth. So we think that the last 1 or 2 chunks at the end of a video stream are probably different as they occur if the user finishes the stream or some other failure happens. The result can be seen in Figure 4.1. Since we use our Transformer in a way to always predict the complete window we already had to take out all samples without a complete prediction window. So to make the other models more comparable we do the same thing and also remove all samples without a complete prediction window. We believe that this does make more sense as our algorithm is not supposed to predict if the stream is ending and also because the prediction does not really make sense if the stream is ending anyway. So for all evaluations we use only the data that has a complete prediction window available.

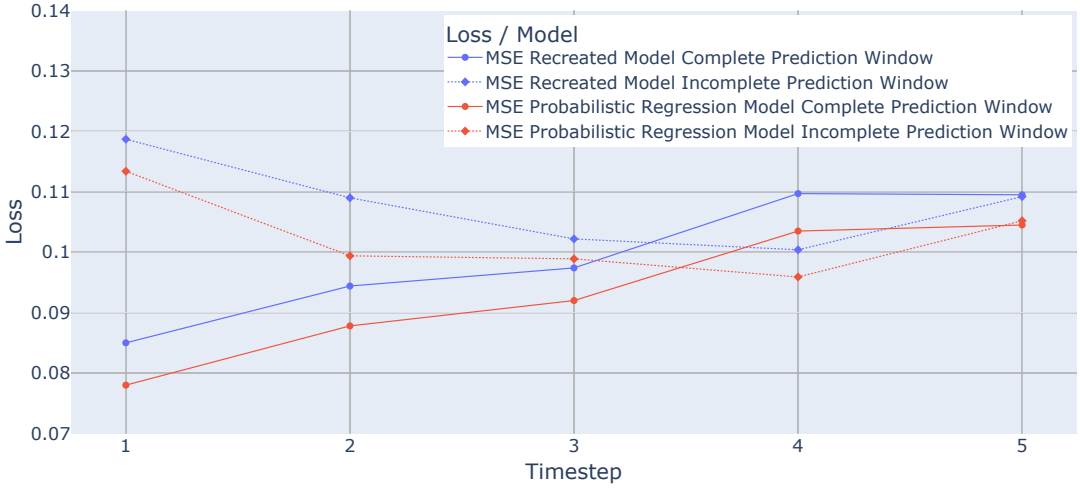


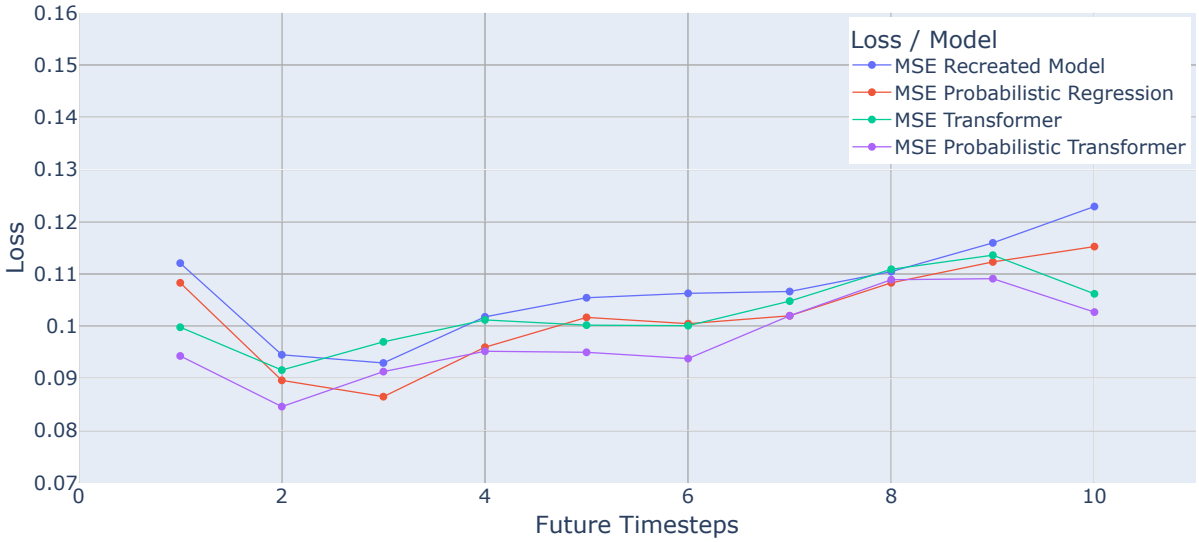
Figure 4.1: MSE for Prediction Window availability

4.3 Larger Prediction Windows

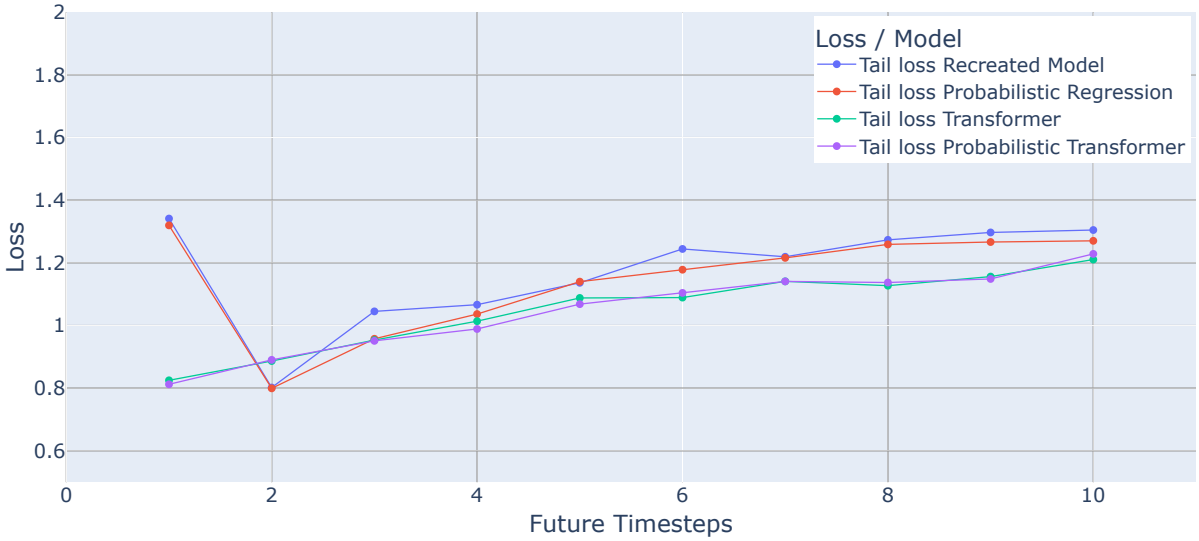
We tried out a bigger prediction window of 10 but found no improvement by using the Transformer.

Data Preparation We prepare the data in the same way as in Section 4.1. We only use the samples that have the complete prediction window available. In this case we only take samples that have a prediction window of 10 available.

Results We would have guessed that the Transformer because of its sequence to sequence nature would be better to predict further into the future but from the graphs in Figure 4.2 we can see that all models have a pretty similar performance evolution over the predicted timesteps.



(a) Mean Square Error



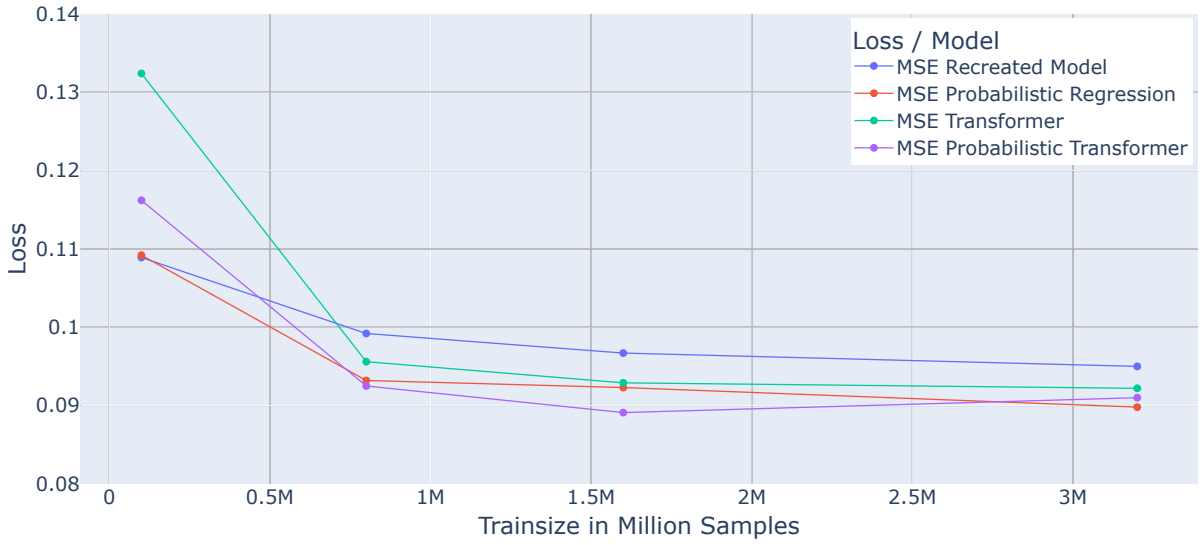
(b) Tail Performance

Figure 4.2: Performance for bigger Prediction Windows

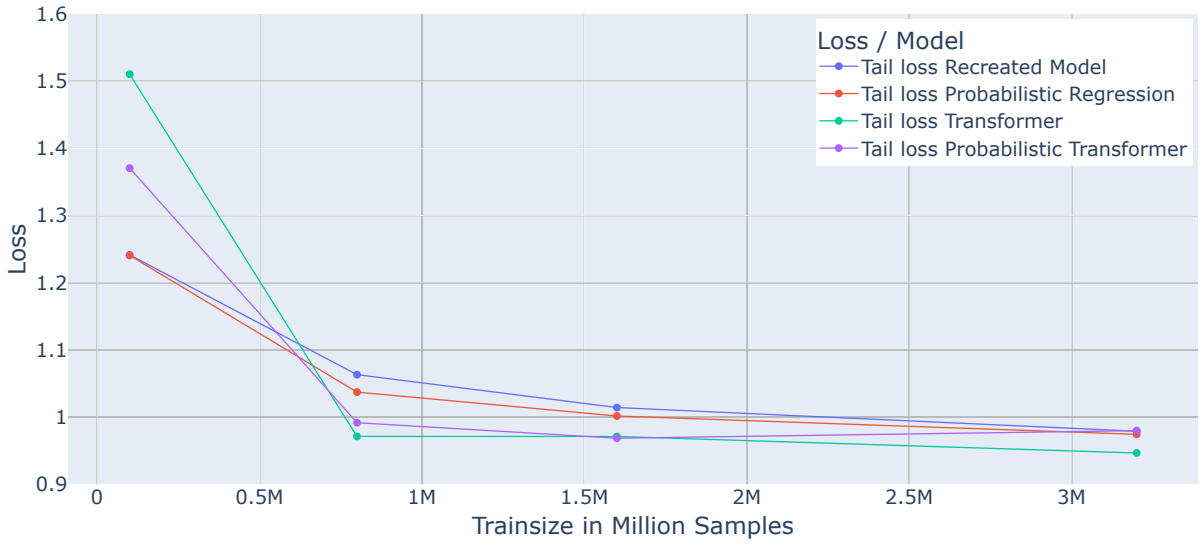
4.4 Training Data Size

We found that using more training samples can improve performance for all models but that it seems to flatten quickly.

Data Preparation We used again the same data from the 27.07.21 which has around 4.3 million samples. We split it up so that we have the same 200k samples as the validation set and then we change the training sample size to 100k, 800k, 1.6 Million and 3.2 Million samples.



(a) Mean Square Error



(b) Tail Performance

Figure 4.3: Training Data Sizes

Results As you can see in Figure 4.3a the Transformer performs much worse than the other models for 100k samples. This is most likely due to the Transformer having more than 10 times the number of parameters as the other two models. Also we can see that all models benefit from having more training data but the returns get pretty small after 800k samples. In Figure 4.3b we can again see the Transformer needing more data than the other models to function properly. But as soon as the Transformer has enough data, its tail performance is slightly better than the other models. For the tail loss the non-Transformer models benefit from more data even beyond 1.6 million samples.

4.5 History Window

Transformers are expected to work well when there is long-term dependency in the data. This does not seem to be the case for the Puffer dataset.

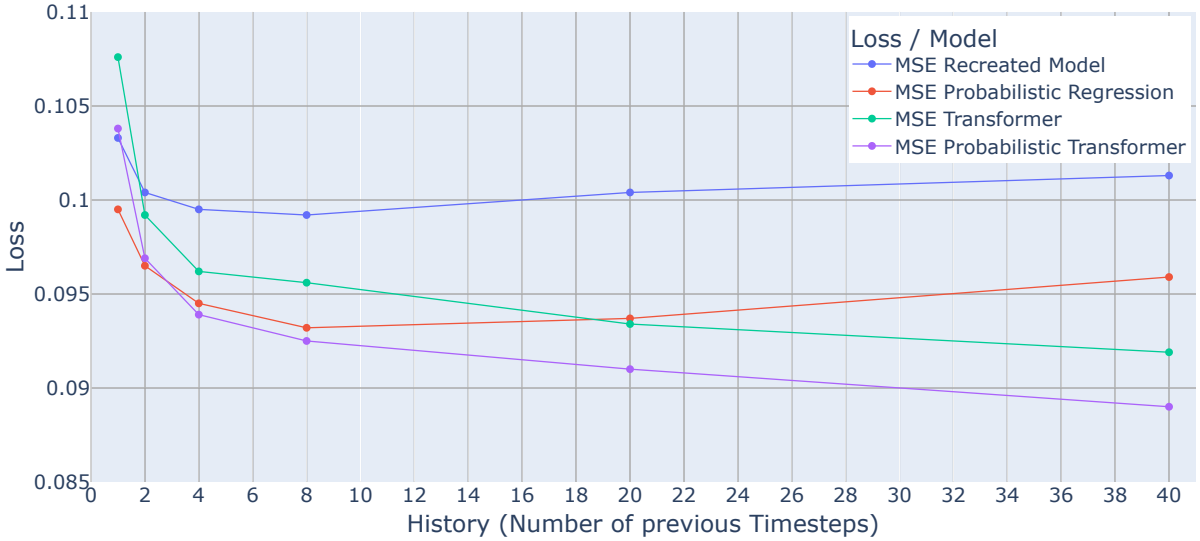
Data Preparation We used different history windows as our input to the model and see how the performance varies with different history windows available to the model. If a sample has not gotten enough history available we will use the same padding as in the original Puffer paper. As a reminder this is just repeating the information of the latest sent video chunk for all previous unavailable timesteps.

Results From Figure 4.4a we can see that the Transformer benefits from a larger history size. But we can also see that the other two models have a sweet spot somewhere between a history of 8 and 20. We think this might be due to the size of those model that they just are not able to handle such big inputs. But the much more important point from this graph is that even for a history of 1 we are only around 15% worse than the best performance which indicates that the most important input data point is the latest timestep. This hints that our prediction task is not really that dependent on the large history window. We can get better performance with the Transformer and larger history sizes but the improvements are rather minimal. For the tail loss in Figure 4.4b we can even see that the performance improvements already plateau at around 20 timesteps of history even for the Transformer. So we do not expect any performance improvements for the tail loss with even more history available to the model. The finding in this experiment also illuminates why we do not see the Transformer notably outperform the other models as we would have guessed before this work. The Transformer needs those long term dependencies to really shine which does not appear to be the case for this dataset. Nevertheless if tail performance is really important, the Transformer can give this small bit of improvement.

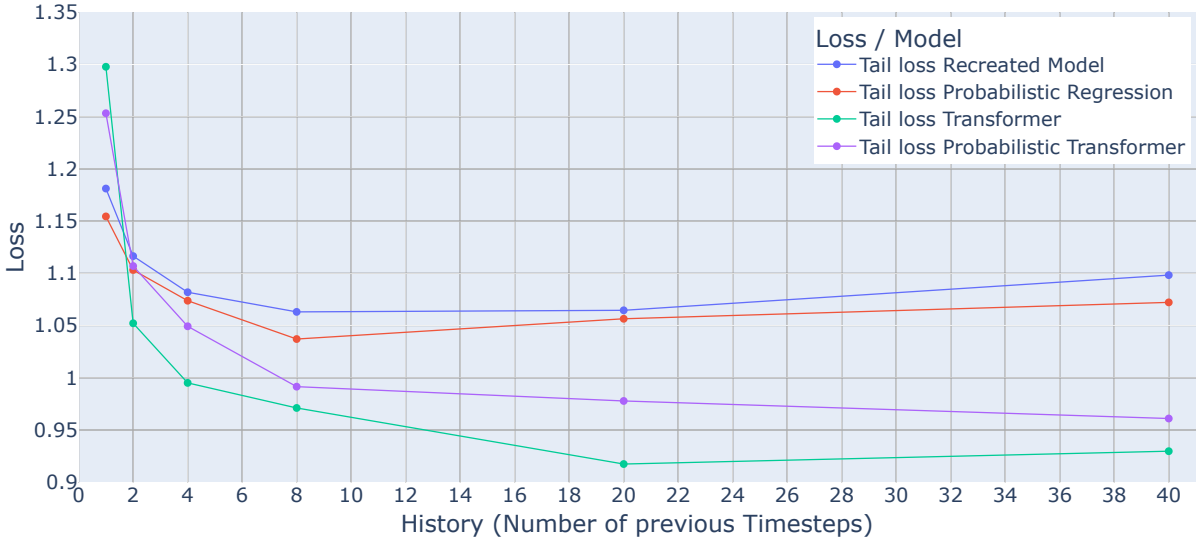
4.6 Influence of the Pretrain Dataset Size

We find that the pretrained Transformer for the probabilistic Transformer does not really benefit from having more than 800k samples as training data.

Data Preparation We first train the standard Transformer with one training dataset size and then freeze those layers and use another training dataset size for the last few probabilistic regression layers.



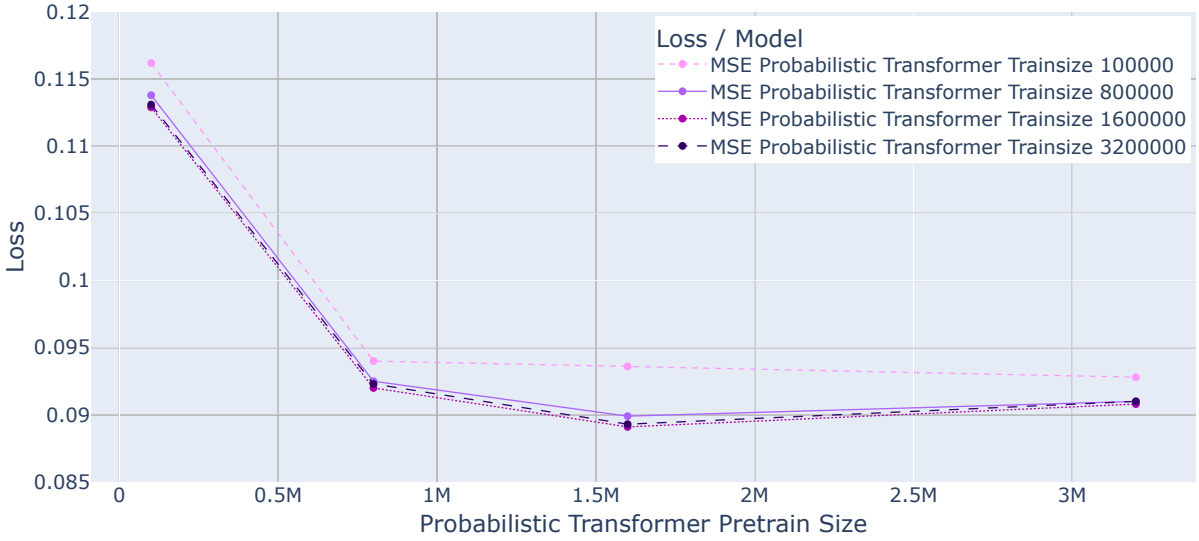
(a) Mean Square Error



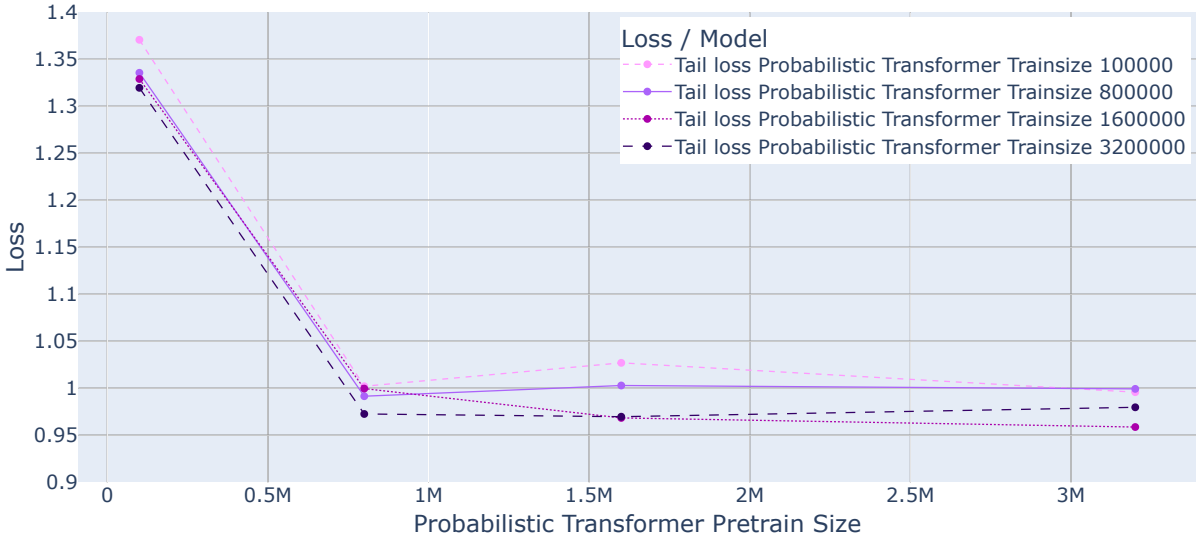
(b) Tail Performance

Figure 4.4: History Input Intervals

Results From Figure 4.5 we can see that beyond a pretrain dataset size of 800k samples we get nearly no benefit in performance. So from this result we would guess that for training the probabilistic Transformer we could get away with a 800k sample pretrained Transformer and then use a bigger datasets just for the training of the last few layers. Pretraining the Transformer with 800k samples takes around 5 hours while it takes approximately 15 hours for 3.2 million samples. So we would save around 10 hours on the training time. The training of the last layers with 3.2 million samples takes around 3.5 hours .



(a) Mean Square Error



(b) Tail Performance

Figure 4.5: Pretrainsize vs Trainsize

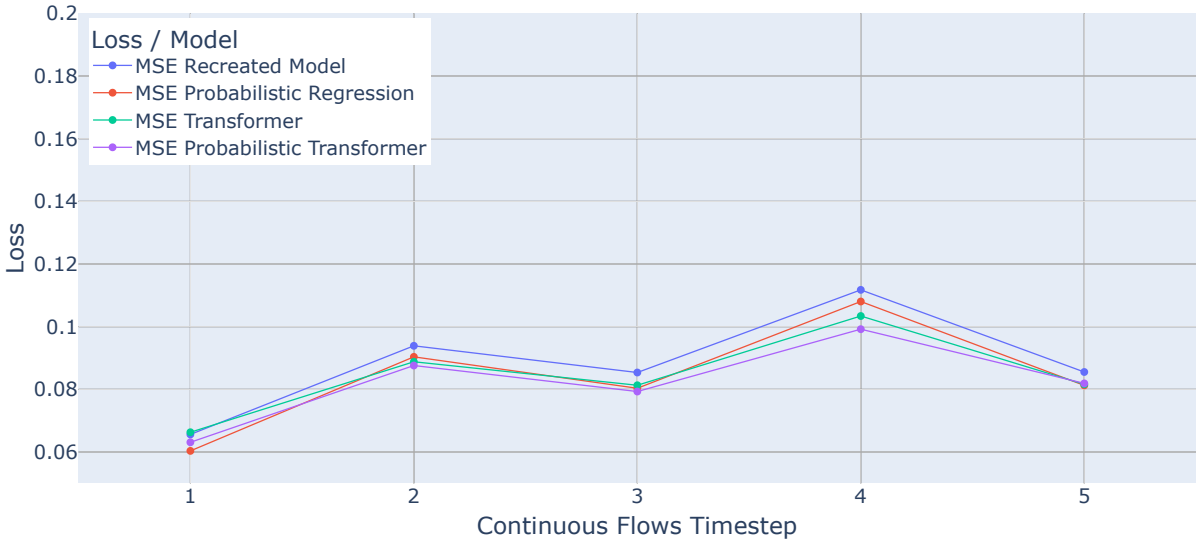
4.7 Start Up vs Running Flows

In this part we see that there exist differences between predicting the first few packets of a flow and predicting transmission times of already running flows.

Data Preparation We split up our samples into two sets. The samples of the first set are part of already running flows with the complete history window available. They are marked as continuous flows. The others without a complete history window available are marked as start up flows. We use the samples from both sets for the training and use as validation sets 20'000 samples once with continuous flow samples and once with start up flow samples. Keep in mind that we can not

compare the absolute results for this experiment with the previous sections as it uses a different validation set.

Convergence Problem We found that the probabilistic regression model has problems to converge so we used upsampling to augment the training dataset. We duplicate the much less numerous short samples (only around 80k) such that we have the same amount of short and long flow samples. If we run the probabilistic regression model with this training set we get a result that falls in line with the other models. So we are confident that the graphs seen below represent inherit properties of the dataset and not something that is model-dependent.

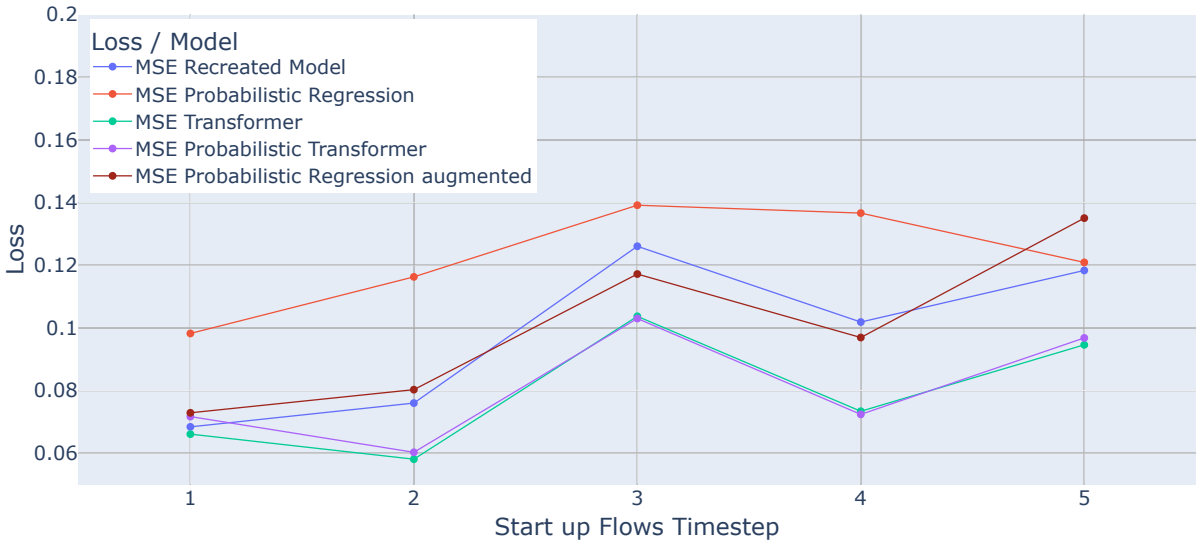


(a) Mean Square Error

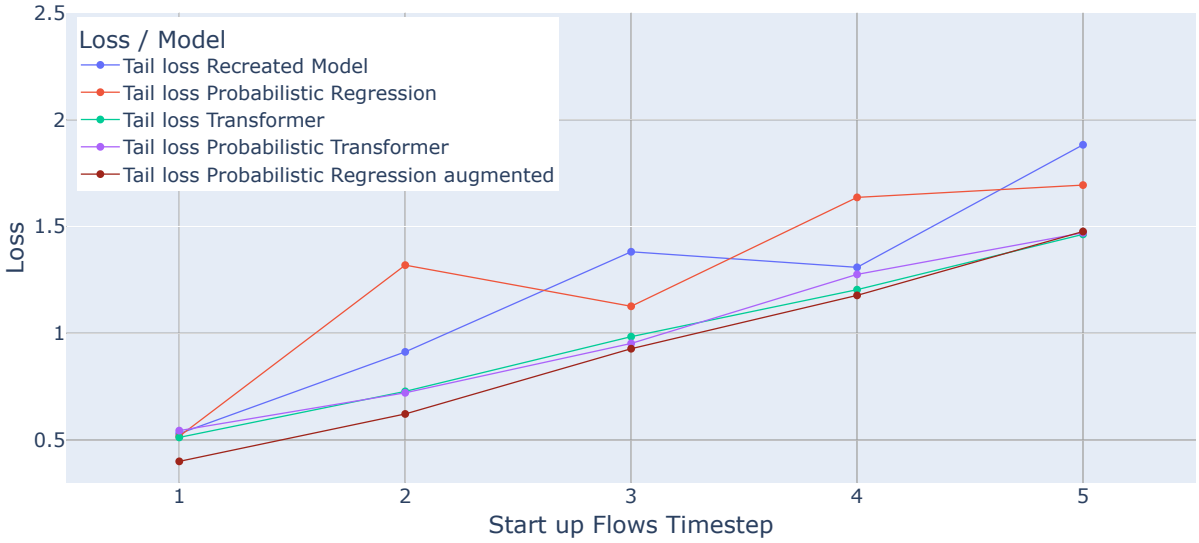
(b) Tail Performance

Figure 4.6: Performance for already running Flows with enough History

Results From the graphs in Figure 4.6 and Figure 4.7 we can see that for the tail loss case it is much easier to predict longer time horizons in already running flows while the error grows nearly linearly for flows in start up. The story is different for the mean where some of the timesteps have even better performance but overall the prediction is not as consistent over the whole prediction window.



(a) Mean Square Error



(b) Tail Performance

Figure 4.7: Performance for starting Flows without enough History

Chapter 5

Outlook

We have two additional ideas that we think could be interesting to look at. One is changing the standard Transformer to a temporal fusion Transformer. The other idea is to investigate if having other/more features might improve the performance.

Temporal Fusion Transformers While working on this paper we found an interesting paper that adapts the Transformer to be better at multi-horizon time series predictions [9]. The model they propose is called a temporal fusion Transformer and is able to handle time series prediction where not all input features are also available for the output. They use both recurrent layers and self-attention layers to get good performance on local and long-term dependencies. Additionally, they have specialized components to select relevant features. It might be interesting to see if such a model would perform better than the standard Transformer for this video streaming task.

Additional Features Another thing that would be interesting to investigate is having more features. For our dataset we only have a hand selected number of 7 features available to us, but there might be other features from the tcp_info struct that might be beneficial to a more complex architecture like the Transformer. It could also be interesting to analyze non-networking related features like server load or number of concurrent streams.

Chapter 6

Summary

In this paper we tested the Transformer model on the video streaming task seen in the Puffer project and found nearly no improvement on the mean performance and a small one on the tail performance. While we are able to outperform the other models on the base task of predicting 5 timesteps into the future, it is only by a small margin that probably does not make a big difference in actual applications. We tried predicting 10 timesteps into the future but also found only marginal benefits when using the Transformer compared to the other models. The models all benefit from having more data available to them but we already observed diminishing returns. We also tried giving the models a bigger history window and found that the Transformer models clearly benefit from it. But the performance benefits are already plateauing beyond a history window of 8. We found that most of the performance can already be reached by only using the last four timesteps as inputs and even when using only a history window of 1 the performance is only marginally worse. These results hint that the video streaming dataset available may not have that big of a time dependency as foreseen. It is also possible that it is more difficult than anticipated to learn these dependencies and that we might need other features to find them. We think that those are the reasons why the Transformer does not clearly outperform the other models as the Transformers strength is to find long term dependencies in the data.

Bibliography

[1] Deep Learning Trends: Top 20 best uses of GPT-3 by OpenAI. https://www.educative.io/blog/top-uses-gpt-3-deep-learning.

[2] PyTorch documentation — PyTorch 1.9.1 documentation. https://pytorch.org/docs/1.9.1/.

[3] PyTorch Lightning — PyTorch Lightning 1.6.0dev documentation. https://pytorch-lightning.readthedocs.io/en/latest/.

[4] Scikit-learn: Machine learning in Python — scikit-learn 1.0.1 documentation. https://scikit-learn.org/stable/.

[5] Scikit-optimize: Sequential model-based optimization in Python — scikit-optimize 0.8.1 documentation. https://scikit-optimize.github.io/stable/.

[6] Bergstra, J., and Bengio, Y. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research 13* (Feb. 2012), 281–305.

[7] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language Models are Few-Shot Learners. *arXiv:2005.14165 [cs]* (July 2020).

[8] Li, S., Jin, X., Xuan, Y., Zhou, X., Chen, W., Wang, Y.-X., and Yan, X. Enhancing the Locality and Breaking the Memory Bottleneck of Transformer on Time Series Forecasting. In *Advances in Neural Information Processing Systems* (2019), vol. 32, Curran Associates, Inc.

[9] Lim, B., Arik, S. O., Loeff, N., and Pfister, T. Temporal Fusion Transformers for Interpretable Multi-horizon Time Series Forecasting. *arXiv:1912.09363 [cs, stat]* (Sept. 2020).

[10] Mohammdi Farsani, R., and Pazouki, E. A Transformer Self-attention Model for Time Series Forecasting. *Journal of Electrical and Computer Engineering Innovations (JECEI) 9*, 1 (Jan. 2021), 1–10.

[11] Popel, M., and Bojar, O. Training Tips for the Transformer Model. *The Prague Bulletin of Mathematical Linguistics 110*, 1 (Apr. 2018), 43–70.

[12] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is All you Need. In *Advances in Neural Information Processing Systems* (2017), vol. 30, Curran Associates, Inc.

[13] Wong, W. What is Teacher Forcing? https://towardsdatascience.com/what-is-teacher-forcing-3da6217fed1c, Oct. 2019.

[14] Wu, N., Green, B., Ben, X., and O'Banion, S. Deep Transformer Models for Time Series Forecasting: The Influenza Prevalence Case. *arXiv:2001.08317 [cs, stat]* (Jan. 2020).

[15] Yan, F. Y., Ayers, H., Zhu, C., Fouladi, S., Hong, J., Zhang, K., Levis, P., and Winstein, K. Learning in situ: A randomized experiment in video streaming. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)* (2020), pp. 495–511.

Appendix A

Multi-GPU Setup

We tried running the models on a multi-GPU setup but we ran into multiple problems. The training strategy we first used was "data parallel" from the pytorch library. This strategy can be used for one node with multiple GPUs. The problem with this strategy is that it does not scale at all above 2 GPUs, so using 4 GPUs was exactly as fast as using 2. The other problem with 2 GPUs was that the model just did not converge as fast and took nearly the same time as 1 GPU as it just took more epochs to reach the same validation performance. We then found out that the "data parallel" strategy is not really supported or well maintained in pytorch which might explain the strange behavior we saw. The other strategy we tried to use was "distributed data parallel" but we could not get it running in a reasonable time as each GPU runs a separate instance of the code which made it difficult to run the validation step with our performance metrics as we needed to collect all data in one GPU to calculate the score. There is no fundamental limitation to getting this to work but there was just no time to investigate it further. If there was more time we would probably rewrite the validation step to safe the results in one file and using one master GPU to calculate the validation scores.