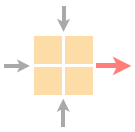
ETH zürich



Networked Systems
ETH Zürich — seit 2015

Advancing Packet-Level Predictions with Transformers

Master Thesis

Author: Siddhant Ray

Tutors: Alexander Dietmüller, Dr. Romain Jacob

Supervisor: Prof. Dr. Laurent Vanbever

Februrary 2022 to August 2022

Abstract

Learning underlying network dynamics from packet-level data has been deemed an extremely difficult task, to the point that it is practically not attempted. While research has shown that machine learning (ML) models can be used to learn behaviour and improve on some specific tasks in the networking domain, these models do not generalize to any other tasks. However, a new ML model called the *Transformer* has shown massive generalization abilities in several fields, where the model is pre-trained on large datasets, in a task agnostic fashion and fine-tuned on smaller datasets for task specific applications, and has become the state-of-the-art architecture for machine learning generalization. We present a new Transformer architecture adapted for the networking domain, the Network Traffic Transformer (NTT), which is designed to learn network dynamics from packet traces. We pre-train our NTT to learn fundamental network dynamics and then, leverage this learnt behaviour to fine-tune to specific network applications in a quick and efficient manner. By learning such dynamics in the network, the NTT can then be used to make more network-aware decisions across applications, make improvements to the same and make the networks of tomorrow, more efficient and reliable.

Contents

Chapter 1

Introduction

Learning fundamental behaviour of network data from packet traces is an extremely hard problem. While machine learning (ML) algorithms have been shown as an efficient way of learning from raw data, adapting such algorithms to the general network domain has been hard, to the point that the community doesn't attempt to do so. In our project, we argue that all is not lost, using some specific machine learning architectures like the Transformer, it is indeed possible to develop methods to learn from such data in a general manner.

1.1 Motivation

Modelling network dynamics is a *sequence modelling* problem. From a sequence of past packets, the goal is to estimate the current state of the network (e.g. Is there congestion? Will the packet be dropped?), following which predict the state's evolution and future traffic's fate. Concretely, we can also decide on which action to take next e.g. should the next packet be put on a different path? Due to the successes in the field of ML and learning from data, it is becoming increasing popular to use such algorithms for solving this modelling problem but the task is notoriously complex. There has been some success in using ML for specific applications in networks, including congestion control[1, 30, 43, 59], video streaming[2, 38, 60], traffic optimization[13], routing[55], flow size prediction[17, 46], MAC protocol optimization[31, 62], and network simulation[64], however a good framework for general purpose learning on network data, still doesn't exist.

Today's ML models trained are trained for specific tasks and do not generalize well; i.e. they often fail to deliver outside of their original training environments[60, 7, 21]. Due to this, generalizing to different tasks is not even considered. Recent work argues that, rather than hoping for generalization, one obtains better results by training in-situ, i.e. using data collected in the deployment environment[60]. Today we tend to design and train models from scratch using model-specific datasets (Figure 1.1, top). This process is arduous, expensive, repetitive and time-consuming. We always redo everything from scratch in the training process, and never make use of common objectives for training. Moreover, the growing resource requirements to even attempt training these models is increasing inequalities in networking research and, ultimately, hindering collective progress.

As ML algorithms (especially certain deep learning architectures) have shown generalization capabilities[41] in other fields, where an initial *pre-training* phase is used to train models on a large dataset, in a task-agnostic manner and then, in a *fine-tuning* phase, the models are refined on smaller task specific datasets. This helps reuse the general pre-trained model across multiple tasks, making it resource and time efficient. This kind of transfer learning or generalization[42]

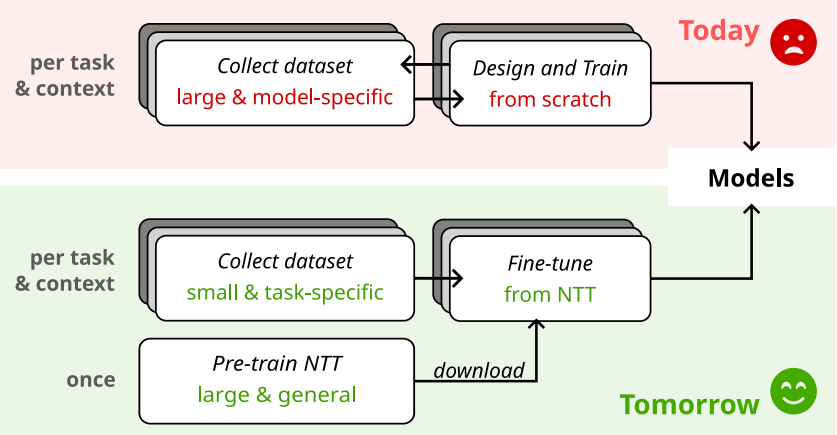


Figure 1.1: Can we collectively learn general network traffic dynamics *once* and focus on task-specific data collecting and learning for *many future models?* Credits: Alexander Dietmüller

is enabled by using the pre-training phase to learn the overall structure in the data, followed by the fine-tuning phase to focus on learning more task-specific features. As long as there is a certain amount of similarity in the data's structure across the pre-training and fine-tuning, this method can have extremely efficient results.

1.2 Tasks, Goals and Challenges

Inspired by ML models which generalize on data in several other fields, it should be possible to design a similar model, in order to have learning and generalization in networking. Even if the networking contexts (topology, network configuration, traffic, etc.) can be very diverse, the underlying dynamics of networks remain essentially the same; e.g. when buffers fill up, queuing disciplines delay or drop packets. These dynamics can be learned with ML and should generalize and it should not be required to re-learn this fundamental behaviour for training a new model every time. Building such a generic network model for network data is challenging, but this effort would benefit the entire community. Starting from such a model, one would only need to collect a small task-specific dataset to fine-tune it (Figure 1.1, bottom), assuming that the pre-trained model generalizes well. This could even allow modelling rare events (e.g. drops after link failures) for which, across the real network traces today, only little data is available due to its less frequent occurrence.

While research shows some generalization in specific networking contexts[30], truly "generic" models, which are able to perform well on a wide range of tasks and networks, remain unavailable. This is caused by the fact that we usually do not train on datasets large enough to allow generalization, we only train on task-specific smaller datasets. Sequence modelling for a long time, has been infeasible even with architectures dedicated to sequence modelling, such as recurrent neural networks (RNNs), as they can only handle short sequence and are inefficient to train[36]. However, a few years ago, a new architecture for sequence modelling was proposed: the *Transformer*[56] and this proved to be ground-breaking for sequence modelling. This architecture is designed to train efficiently, enabling learning from massive datasets and unprecedented generalization. In a *pre-training phase*, the transformer learns sequential "structures", e.g. the structure of a language from a large corpus of texts. Then, in a much quicker *fine-tuning phase*, the final stages of the model are adapted to a specific prediction task (e.g. text sentiment analysis). Today, Transformers are among

the state-of-the-art in natural language processing (NLP[52]) and computer vision (CV[25]).

The generalization power of the Transformer, stems from its ability to learn "contextual information", using context from the neighbouring elements in a sequence, for a given element in the same sequence[16].[1] We can draw parallels between networking and NLP. In isolation, packet metadata (headers, etc.) provides limited insights into the network state, we also need the *context*, i.e. which we can get from the recent packet history.[2] Based on these parallel, we propose that a Transformer based architecture can also be design to generalise on network packet data.

Naively transposing NLP or CV transformers to networking fails, as the fundamental structure and biases[39] in the data are different. Generalizing on complex interactions in networks is not a trivial problem. We expect the following challenges for our Transformer design.

- How do we adapt Transformers for learning on networking data?
- How do we assemble a dataset large and diverse enough to allow useful generalization?
- Which pre-training task would allow the model to generalize, and how far can we push generalization?
- How to we scale such a Transformer to arbitrarily large amounts of network data from extremely diverse environments?

1.3 Overview

We present in our thesis, a Network Traffic Transformer (NTT), which servers a first step, to design a Transformer model for learning on network packet data. We outline the following main technical contributions in this thesis:

- We present the required background on Transformers, which gives directions to our design, in Chapter 2.
- We present the detailed architectural design ideas behind our proof-of-concept NTT in Chapter 3.
- We present a detailed evaluation on pre-training and fine-tuning our first NTT models in Chapter 4.
- We present several future research directions which can be used to improve our NTT in Chapter 5.
- We summarise our work and provide some concluding remarks in Chapter 6.
- Supplementary technical details and supporting results are presented in Appendix A, B and C.

A part of the work conducted during this thesis has been submitted as the following paper[18] to HotNets '22, and hence, we have some parts of the thesis have been built upon work done in writing the paper.

[1] Consider the word *left* in two different contexts: I *left* my book on the table. Turn *left* at the next crossing. The transformer outputs for the word *left* are different for each sequence as they encode the word's context.

[2] Increasing latency over history indicates congestion.

Chapter 2

Background and Related Work

A large part of the work undertaken during this project requires a deep understanding on how a particular deep learning architecture, the Transformer works. In this section, we will cover some of the required background and insights drawn from the Transformer architecture which were needed to model and solve our problem of prediction on packet data. We also present adaptations of the Transformer architecture to solve problems in several fields such as NLP/CV and relevant ideas which could be adapted to our tasks.

2.1 Background on Transformers

2.1.1 Sequence modelling with attention

Transformers are built around the *attention mechanism*, which maps an input sequence to an output sequence of the same length. Every output encodes its own information and its context, ie information from related elements in the sequence, regardless of how far they are apart. The process involves the scalar multiplication of the input feature matrix attention matrix as a dot product operation, which allows the deep neural network to focus on certain parts of the sequence at a time, based on the values of the attention weight matrix. This allows the network to attend to parts of the sequence in parallel, rather than in sequence, which allows highly efficient computation. Also, as the attention weights are learnable parameters for the network, the Transformer over time, learns to choose the best weights which allow optimum learning of structure within the sequence of data. Computing attention is efficient as all elements in the sequence can be processed in parallel with matrix operations that are highly optimised on most hardware. These properties have made Transformer based neural networks the state-of-the-art solution for solving many sequence modelling tasks. We refer to an excellent illustrated guide to the Transformer here[3].

While attention originated as an improvement to Recurrent Neural Networks(RNNs), it was soon realised that the mechanism could replace them entirely.[56] RNNs were the initial state-of-the art deep learning architectures in sequence modelling problems, however they suffer from several issues. Training RNNs is usually limited by one or all of the following problems:

1. RNNs are not computationally efficient for training long sequences, as they require n sequential operations to learn a sequence, n being the length of the given sequence. This makes the training process extremely slow.

2. RNNs suffer from the problem of *vanishing gradients* i.e. as elements in the input need to be processed in a sequence over time, the gradients used by the optimiser[48] for the elements at

the end of very long sequences, become extremely small and numerically unstable to converge to the desired value.

3. RNNs struggle to learn *long-term dependencies*, i.e. learning relations between elements far apart in the sequence is challenging.

We present an excellent summary of the complexities and differences between several deep learning architectures which have attempted to solve the sequence modelling problem in Table 2.1.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Transformer(Self-Attention)	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent NN	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional NN	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(\log_k n)$
Restricted(Self-Attention)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Table 2.1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighbourhood in restricted self-attention. SOURCE: Original paper[56]

Augmenting RNNs with attention solves some of these issues[6], and replacing them with *Transfomers* has shown to solve all these problems. The authors propose an architecture for translation tasks that contains:

- a learnable *embedding* layer mapping words to vectors
- a *transformer encoder* encoding the input sequence
- a *transformer decoder* generating an output sequence based on the encoded input

Each transformer block alternates between attention and linear layers, ie between encoding context and refining features. The attention mechanism helps learn "context rich" representations of every element, mapping relevant information from the surrounding elements which surround it and the linear layers help map this learn information into a form useful for downstream prediction. Figure 2.1 shows details of the original Transformer architecture and the functions of its layers.

2.1.2 Pre-training and fine-tuning

Due to the highly efficient and parallelizable nature of Transformers, they can be widely used a variety of tasks based on the principle of *transfer learning*. The most common strategy for that is to use the architecture for two phases, *pre-training* and *fine-tuning*. Inspired by the original Transformers success on the task of language translation, the use of Transformers has become ubiquitous in solving NLP problems. We present one such state of the the art NLP Transformer model, called BERT[16], one of the most widely used transformer models today. While the original Transformer had both an encoder and decoder with attention, BERT uses only the transformer encoder followed by a small and replaceable decoder. This decoder is usually a set of linear layers and acts as a multilayer perceptron (MLP); and is usually called the 'MLP head' in the deep learning community. The principle of transfer-learning works for BERT as follows:

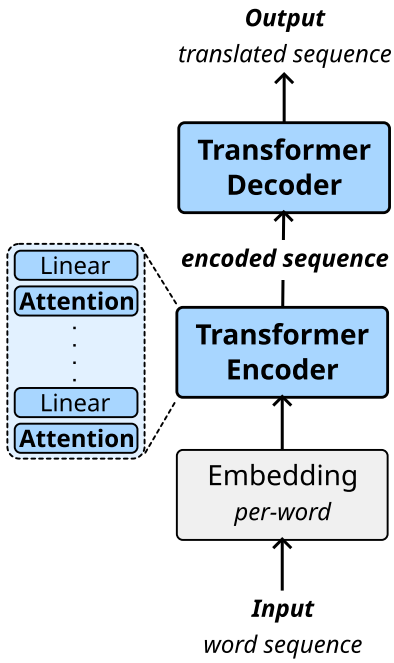


Figure 2.1: Original Transformer Architecture, Credits: Alexander Dietmüller

- In the first step, BERT is *pre-trained* with a task that requires learning the underlying language structure. Linguists' research has shown that a Masked Language Model is optimal for deep learning models for learning structure in natural languages.[57] Concretely, a fraction of words in the input sequence is masked out (15% in the original model), and the decoder is tasked to predict the original words from the encoded input sequence. BERT is used to generate contextual encodings of, which is only possible due to the bi-directionality of the attention mechanism in BERT. This allows the model to infer the context of the word from both sides in a given sentence, which was not possible earlier when elements in a sequence were only processed in a given order.[1]

- In the second step, the unique pre-trained model can be fine-tuned to many different tasks by replacing the small decoder with task-specific ones, eg language understanding, question answering, or text generation. The fine-tuning process involves resumption of learning from the saved weights of the pre-training phase, but for a new task or learning in a new environment. The new model has already learned to encode a general language context and only needs to learn to extract the task-relevant information from this context.This requires far less data compared to starting from scratch and makes the pre-training process faster.

  Furthermore, BERTs pre-training step is unsupervised/self-supervised, i.e. it requires only "cheap" unlabelled data and no labelled signal from the data a target value. As procuring labelled data is harder, this problem is mitigated by having a pre-training phase on "generic" data and then using "expensive" labeled data, eg for text classification, during the fine-tuning phase. Figure 2.2 shows the details of BERT's pre-training and fine-tuning phase.

[1]BERT is pre-trained from text corpora with several billion words and fine-tuned with ∼100 thousand examples per task.

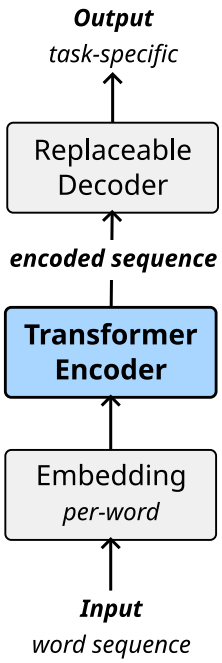


Figure 2.2: Original BERT Architecture, Credits: Alexander Dietmüller

Another extremely useful feature of Transformer is its generalization ability, which is possible due to its highly efficient parallelization during training, which in turn helps transfer of knowledge to a variety of tasks during the process of fine-tuning. The OpenAI GPT-3[11] model that investigates few-shot learning with only a pre-trained model. As it is just a pre-trained model, it does not outperform fine-tuned models on specific tasks. However, GPT-3 delivers impressive performance on various tasks, including translation, determining the sentiment of tweets, or generating completely novel text. As per requirement, it can also be fine-tuned on specific tasks, which showcases further, the generalization power of transformers.

2.1.3 Vision transformers

Following their success in NLP, the use of Transformers were explored as a possibility to learn structural information in other kinds of data. One such field where they gained a lot of traction in in the field of CV. However adapting the use of Transformers to vision tasks came with a few major challenges:

- Image data does not have a natural sequence, as they are a spatial collection of pixels.
- Learning encodings at the pixel level for a large number of images, proved to be too fine-grained to scale to big datasets.
- The Masked Language Model theory couldn't be efficiently transferred to the context of learning structure in images, as the relationship between the units (pixels) does not follow the same logic as in natural languages.

To solve this problems, the authors of the Vision Transformer(ViT)[19], came up with multiple ideas to solve each of these problems, in order to have the data in a form, on which a Transformer could be efficiently trained.

- *Serialize and aggregate the data:* As image data does not have a natural sequence structure, such a structure was artificially introduced. Every image was split at the pixel level and aggregated into patches of dimension 16×16 and each of these patches became a member of the new input "sequence". As Transformers scale quadratically with increasing sequence size 2.1, this also solved the problem of efficiency and that encodings at the pixel level are too fine-grained to be useful. The embedding and transformer layers were then applied to the resulting sequence of patches, using an architecture similar to BERT.

- *Domain Specific Pre-training:* At the heart, most CV problems have very similar training and inference objectives, one of the most common being image classification, which makes most of the vision tasks similar in objective, differing only in environment. This made classification a much more suited task for image data pre-training as opposed to masking and reconstruction. This was exploited by the ViT and both the pre-training and fine-tuning was done with the objective of classification, with only a change in environment between the stages. This also meant that understanding structure in image data, not only required information from the neighbouring patches but also from the whole image, which was possible by using all the encoded patches for classification. We present the details of ViT's pre-training and fine-tuning in Figure 2.3.

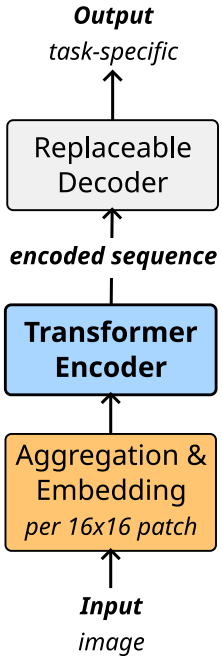


Figure 2.3: Original ViT Architecture, Credits: Alexander Dietmüller

Finally, ViT's authors also observe that domain-specific architectures that implicitly encode structure, like convolutional networks (CNNs) work extremely well for image datasets and actually result in better performance on small datasets. However, given enough pre-training data, learning sequences with attention beats architectures that encode structure implicitly, making the process a tradeoff between the utility and the amount of resources required for pre-training. Later research in the field has also shown advancements in vision based transformers which use a masked reconstruction approach[26], such details however are beyond the scope of this section.

2.2 Related Work

The problem of learning networks dynamics from packet data has been deemed a complex "lost cause", and not a lot of the research community has made much direct effort in this direction. The idea of using large pre-trained machine learning models to learn from abstract network traces, has thus not been explored much. However, application specific adaptions of ML architectures have been used with varying amount of success on network data, and we present some of these efforts and successes here, which helped provide direction to our own ideas and thoughts for this project.

In MimicNet[64], the authors show that they can provide users with the familiar abstraction of a packet-level simulation for a portion of the network while leveraging redundancy and recent advances in machine learning to quickly and accurately approximate portions of the network that are not directly visible. In Puffer[60], they use supervised learning in situ, with data from the real deployment environment, to train a probabilistic predictor of upcoming chunk transmission times for video streaming. The authors of Aurora[30] show that casting congestion control as a reinforcement learning (RL) problem enables training deep network policies that capture intricate patterns in data traffic and network conditions but also claim that fairness, safety, and generalization, are not trivial to address within conventional RL formalism. Pensieve[38] trains a neural network model that selects bitrates for future video chunks based on observations collected by client video players and it learns to make ABR decisions solely through observations of the resulting performance of past decisions. Finally, in NeuroPlan[65] they propose to use a graph neural network (GNN) combined with a novel domain-specific node-link transformation for state encoding and following that, leverage a two-stage hybrid approach that first uses deep RL to prune the search space and then uses an ILP solver to find the optimal solution for the network planning problem.

Chapter 3

Design

Inspired by the success of Transformer architectures on learning sequence modelling tasks in NLP and CV, we propose that we can adapt the same to learn dynamics from network packet data due to the underlying similarity of sequential data. In this chapter, we first present some insights on the data we use for learning, following which, we present the ideas and design choices behind a proposed Transformer architecture which can be used for the same.

3.1 Pre-training Dataset

The key requirement for using any deep learning architecture to model and solve a learning problem is the availability good quality training data, which contains enough variation in its samples in order to capture the underlying distribution effectively. We need enough training data so as the model can learn effectively and not underfit on the training samples. At the same time, we want enough variation in the training data so as to not overfit on a certain subset of features, in order to achieve good generalization benefits from the training process. Given enough data of the right kind, deep learning models are known to exhibit extremely powerful generalizing behaviour[41].

Acquiring the network packet data to train our proposed NTT model was an initial challenge . A large number of real network traces are sparse (e.g. only 1 hour of data every month) and this is not optimal for training deep learning models. Other kinds of network data while not sparse, is not readily available. Data centres today experience millions of flows per minute across varying levels of granularity and while these can be leveraged to form an ensemble of training data for the specific tasks, the data is usually limited to certain organizations and is not available easily for general use. In order tackle this problem, we use data from network simulations, which allow us to control the ground truth and better understand the performance of the model, while generating data as per our requirement. Latest research has also shown that network simulations and emulations learnt from part of a network and replicated across the varying smaller clusters in the network, can work as almost as well as training on the real network traces, if not the same. Authors in MimicNet[64] have shown the effectiveness and utility of such techniques.

For our task, we use the Network Simulator 3(ns3)[47] to generate training and testing data for our NTT. To ensure we capture enough real-world network dynamics, we generate our simulation data from a collection of real world message size distributions, which have been acquired from actual data centres. Given the distribution of these message sizes across multiple workloads, we use ns3 to generate traffic traces from them with varying levels of load.[40].

We begin with a simplified scenario of a single-link network topology which connects two hosts, via two switches as shown in Figure 3.1, where we show the delay Cumulative Density Function

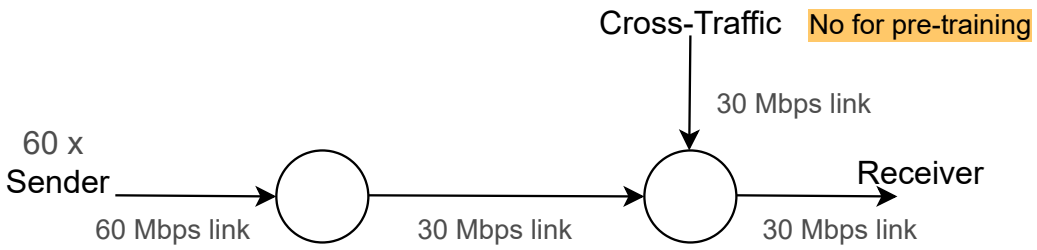
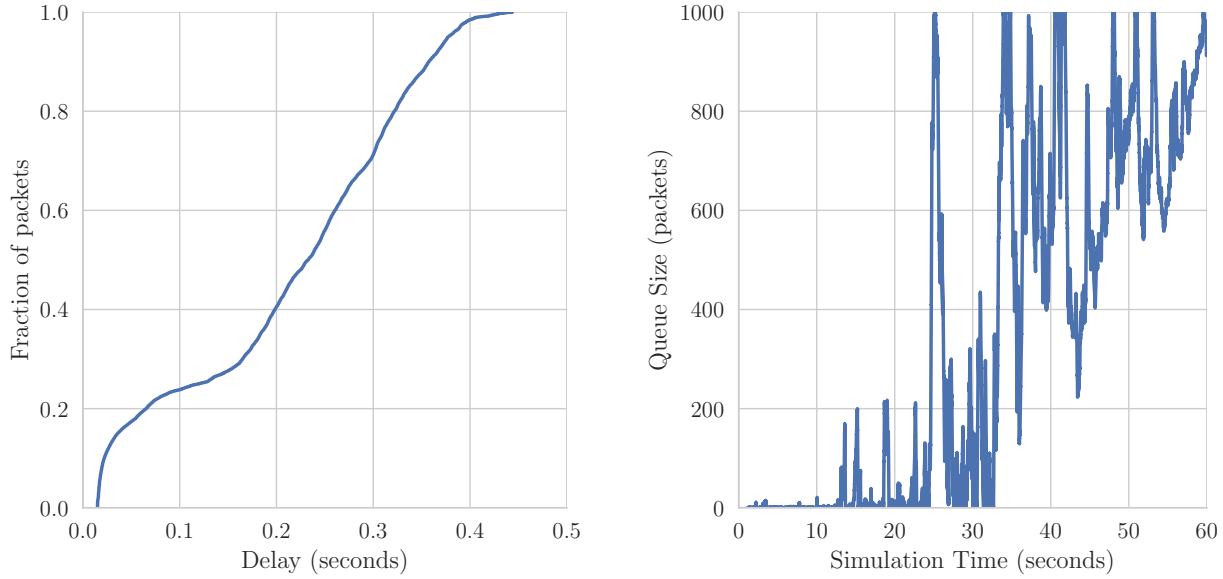


Figure 3.1: Initial topology for data generation

(CDF) and the bottleneck queue behaviour. For the pre-training dataset, each of 60 senders generates $1Mbps$ of messages from the above mentioned distributions. They send messages over a bottleneck link with $30Mbps$ bandwidth and a queue size of 1000 packets. We run 10 simulations for 60 seconds, each with randomized application start times in a $1-50$ second window. This also ensures none of the applications are too short to actually affect the overall dynamics of the network. The randomization of the start times ensures complex interactions between different flows, which lead to significant variation in dynamics of the network. This helps make the data for the pre-training capture a large, generic underlying distribution, which is hugely beneficial for fine-tuning on specific tasks later. This pre-training dataset contains about 1.2 million packets. There is a possibility of introducing cross-traffic using a second source of disturbance on the second switch, however we turn it off for generating pre-training data, and try to learn general dynamics from a single bottleneck in the topology.



(a) Delay CDF, single run

(b) Queue profile on bottleneck queue

Figure 3.2: Distribution plots on pre-training data

It is important to ensure that we do not have too much average behaviour in the generated data, else it will prevent the deep learning model from capturing the tail behaviour of the distributions. While in a lot of fields, learning average behaviour with deep learning is more or less enough, this

idea fails with network data. In networks, worst-case scenarios can, even if occasionally, lead to significant network outages and data losses. In such an environment, it is crucial to understand the tail behaviour of the traffic distributions across various network applications, in order to truly learn the network characteristics and use the information gained to improve the performance of the network. We ensure that the features required for our training, end to end delay and packet size(we present the choice behind this selection in greater detail in Section 3.2.2) have enough variation in our training data. We also ensure that the state of the queue on our bottleneck varies throughout the simulation, i.e. it is neither always full and nor always empty, which introduces much more variation in the underlying dynamics of the training data. We refer to these distributions in Figure 3.2.

3.2 Network Traffic Transformer (NTT)

We present our proof of concept Network Traffic Transformer(NTT) as a first Transformer model which is trained to learn sequence structure in network packet data and capture the underlying dynamics of the same, and generalize to new tasks based on the transfer learning principles from deep learning . However, learning the structure from network packet data can be extremely challenging. We face several major problems which need to be addressed.

1. *Complexity of Data:* Network packet data is much more complex as compared to data from images(pixels) or from sentences(words). Packet data comes with several layers of hierarchy in terms of headers, protocol stack etc. We need to devise an effective way to choose features for learning, which captures all of the required information. At the same time, we need to ensure that we don't have redundancy in choosing the features, eg some fields across protocols and headers may present the same information in multiple ways. Choosing the features from packets which are the best learning becomes an incredibly hard task.

2. *Time-Series Data:* Neither data in CV nor NLP follows a time series structure, but in network packet data it does, which poses a big challenge. The fate of a packet in the the data depends often on a packet much further in the past, which leads to the need to learn from extremely long sequences of data. As discussed, Transformers scale quadratically with in increasing sequence size, hence learning from long sequences is a challenge. At the same time, the fate of a packet may be affected much more by the more recent packets, which leads to the challenge of effectively learning from both short term and long term dynamics.

3. *Training tasks:* The structure of data in NLP and CV is well understood by humans and it makes it easier to design pre-training and fine-tuning training objectives for these problems. Network packet data is extremely abstract and it significantly harder for us to parse this information and understand it. This makes developing such training objectives on network packet data a bigger challenge. We need to effectively design these in order that the sequence structure can be learnt to the best extent and in the most efficient manner possible.

3.2.1 Sequence definition

An initial challenge which came up in our sequence modelling problem was choosing an appropriate sequence definition for our training and evaluation samples. In using Transformers for NLP, the concept of sequence is well defined in terms of either sentences or paragraphs, which come from

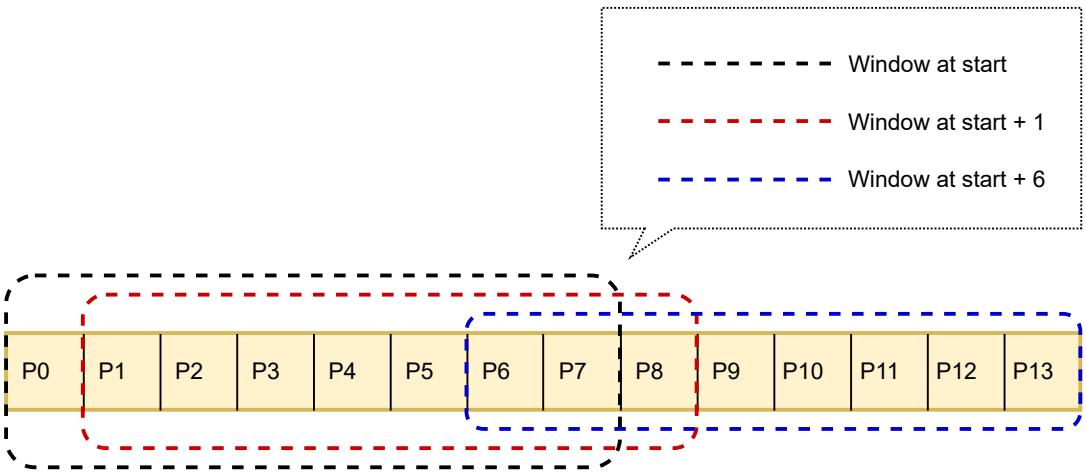


Figure 3.3: Sliding window for sequence selection. Here P0, P1 etc. are packet-level features used as input to the NTT.

the structure of most languages itself[1] In using Transformers for CV, concept of sequence was artificially constructed using the concept of splitting the image into fixed size patches, which has been covered in detail in Section 2.1.3. In our case, we have as pre-training data, a time-series dataset, in increasing order of timestamp from the first packet in the network trace.

As our network packet data sequence does not have natural delimiters[2], we introduce a concept of a sliding window over our packets, in order to create a fixed sequence length, which will then we provided as input to our NTT. We choose a *one-step* sliding window of 1024 packets,, which roughly matches the size of the queue on the bottleneck switch, over our entire network trace, and each of these windows becomes a single sequence for us. At every step, we shift the sliding window forward by a single step and we illustrate this in Figure 3.3. We expect that this will capture network behaviour when the queue is full and we hypothesise such a condition to be important for learning the dynamics. The concept of sliding windows has also been successfully used in cases on NLP data, which doesn't have the necessary delimiters in the data itself[8], which makes such a design choice a natural starting point in our case. Our sliding window becomes a sequence of elements, which each individual element containing a certain number of features which are required for learning on our training objective.

Our pre-training data contains traces from multiple runs of simulation, in order to increase the richness of our pre-training data and enable us to learn variation in dynamics and generalise to a wide variety of tasks. These simulations represent network measurements taken at different points of time and it is important that our sliding windows always contain elements from the same run of simulation. Since data from different simulation represents independent measurements, we ensure that we construct our sliding windows independently over each run of the simulation data, and then use the data from the combination of these sliding windows as our training samples.

3.2.2 Feature selection for the NTT

Packets carry a lot of information that could be used as model features, e.g. all header fields, payload size etc. Today, we typically use networking domain knowledge to manually extract and aggregate

[1]We talk about structure in Indo-European languages here which is not universally true for all language families.
[2]No equivalent of punctuation marks like "." in NLP.

features, then feed them into off-the-shelf standard ML architectures. This is done repeatedly for every task required, we always choose features which will be the best choice for learning our task objective and then, train a new ML model from scratch for every application. We argue this is sub-optimal for two reasons:

- First, we always tailor these features to a specific task and dataset, which limits generalization. This process is redundant and never lets us benefit from previous training experiences and results.
- Second, since the features are not learned from data, they may end up sub-optimal for the task. Choosing the right features for training an ML model is incredibly hard. The only foolproof method is to perform an exhaustive search over all possible combinations of features, and then choose the best performing set/subset of features. Several methods which do this are GridSearchCV and KFoldCV[45] but these methods are extremely slow and resource intensive.

We propose to let the model learn useful features from raw data. To learn traffic dynamics from a sequence of packets, we must provide the model with information about the packets as well as their fate in the network. Since we do not want to define a priori how important the individual pieces of information are, we feed them all into a first embedding layer (see Figure 3.4). The layer is applied to every packet separately. We also tried variations with using a single embedding for the entire sequence and no embeddings at all, both of which performed extremely poorly. The embedding layer itself is a linear projection layer, which helps increase the representation of the information present in the selected features. Also, as these embeddings are learnable parameters in our NTT, over the training process, the model can learn which is the best representation for our input features, from the point of view of feeding information into the transformer encoder.

We argue that we need both information about the packet state and the network state to capture the overall network dynamics. In our proof-of-concept NTT, we use minimal packet and network information and argue that the selected features have enough information to model the network dynamics which is needed for our training objectives.

- *relative timestamp:* Transformer architectures process items in a sequence in parallel, hence they need some positional information in the input in order to understand the relative order of the elements in the sequence. These positional embeddings can be provided as fixed absolute[56] or relative[50] or as learned positional embeddings[22]. Since our packet data is a time-series sequence, we use the relative timestamp to our first packet as a direct feature for our positional encoding information. The relative timestamp is part of the features passed through the embedding layer, and thus becomes a learnable value for a positional encoding.
- *packet size:* The packet size is used as a feature as it determines the fate of the packet from the packet state. The size of the packet affects the number of packets which can be accommodated in the queue buffer, e.g. larger packets means less packets are present in the queue at the particular point in time. This helps in learning the dynamics of the fate of the packet which is directly affected by the state of other packets.
- *end-to-end delay:* The end to end delay in our setup includes both the queueing delay of the packet, along with the path delay. This delay reflects the state of the network at any given point of time, as the delay is caused by the complex interaction between packets and also reflects effects of tangible networks conditions such as congestion, packet drops etc.

These enable learning embeddings with temporal (evolution of delays over time) and spatial (impact of packet size on the delay) patterns. At this point, the clear hypothesis is that these 3

features will be needed in some form at least, for the NTT to learn and generalize on the network dynamics. It may also happen that for future versions of the NTT, when fine-tuned on complex topologies, may require additional features to distinguish between differences in the data, in terms of path travelled, subnets they belong to etc. In Chapter 4, we will investigate need for such further features and discuss the challenge of embedding more such information in Chapter 5.

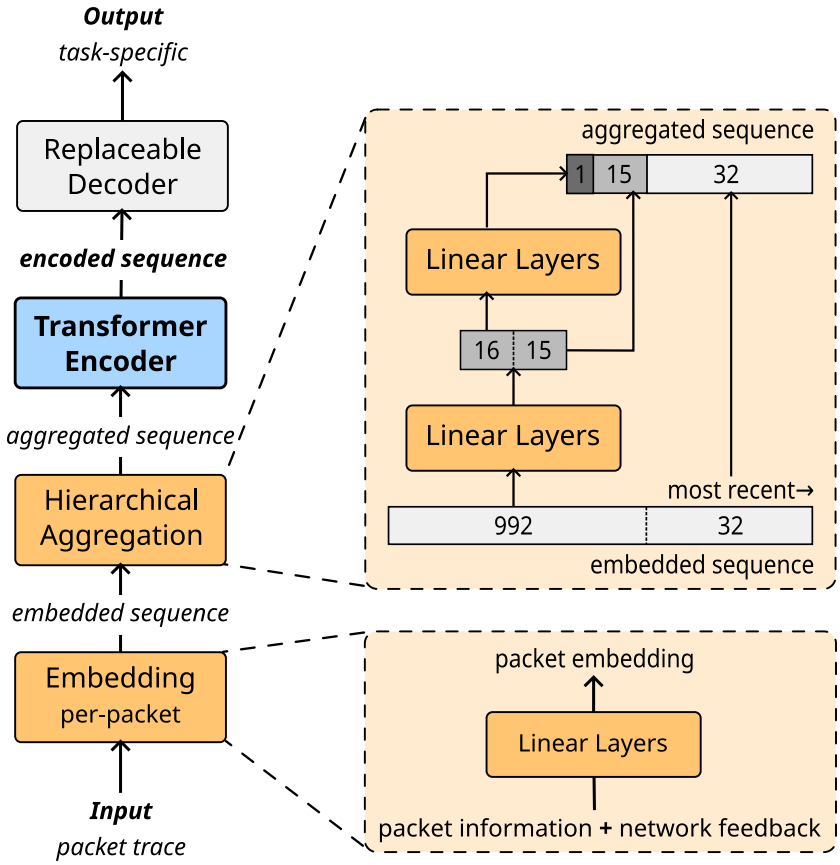


Figure 3.4: The Network Traffic Transformer (NTT) contains three main stages: an embedding layer, an aggregation layer, and a transformer encoder. It outputs a context-rich encoded sequence that is fed into a task-specific decoder. Credits: Alexander Dietmüller

3.2.3 Learning packet aggregation

As we hope to learn network dynamics, packet sequences must be sufficiently long to capture both the short-term and the long term effects effectively. In Section 3.2.1, we discuss our reasoning behind choosing 1024 packets as an appropriate sequence size choice for our initial learning objective. However as the training time of Transformers scales quadratically with the sequence length, we face practical limitations, which we have discussed in Table 2.1. In other fields this is not a a problem, for e.g. in NLP, the maximum sequence size in BERT is limited to 512 elements only[16]. Network packet data is extremely fine-grained and depending on the extent of effects we want to capture in our sequence, we may require extremely large sequences of packets, sometimes even in the order of 10000s or more.

We address this problem by using multi-timescale aggregation in Figure 3.4. We aggregate a long packet sequence into a shorter one while letting the model learn to extract relevant historical information from the aggregated sub-sequence. Coming up with an aggregation scheme which works universally is hard. While coming up with always a fixed aggregation scheme over a constant number of packets is the easiest and probable first choice[3] , we argue that this is suboptimal for two reasons.

- First, it is not easy to choose such a fixed number, as this might be extremely dependent on the future tasks we want to use our NTT for.
- Secondly, we hypothesise that due to the nature of structure of packet data, it is more useful to have representation of both aggregated and individual packets. A fixed aggregation size would mean giving up packet level details completely, which may cause loss in more fine-grained details.[4]

In order to achieve this, we keep the most recent packets without aggregation and the longer traffic is in the past, the more we aggregate, as packet-level details become less relevant to predict the current traffic dynamics. We do this on the argument that most recent information at the packet level is more useful for predict next element's behaviour, whereas individual packets much further in the past matter less and less depending on to which extent in the past they are present.

In our initial NTT experiments, we reduce our initial sequence length of 1024, using a two layer aggregation scheme, which we aggregate into 48 elements. The right-hand side of Figure 3.4 shows the aggregation details. We present further details in Table 3.1.

Packets in Initial Sequence	# times aggregated	Position in Final Sequence
Most recent 32	None	16-48
512-992	Once	1-16
1-512	Twice	1

Table 3.1: Multi-level packet aggregating in the sequence of 1024. The most recent 32 packets are not aggregated, the 480 packets between 512 and 992 are aggregated once, and the packets from 1-512 are aggregated twice. All packets are individually embedded before aggregation.

Our multi-timescale aggregation can easily be adapted to a larger sizes of past history without sacrificing recent packet details. The initial choice of numbers to map our initial sequence length to our aggregated sequence are slightly arbitrary and do not have a sound theoretical basis yet. We also do not know if matching the packet sequence size to the number of in-flight packets (from the current buffer sizes which give us 1024 packets) is the best definition of sequence, but it is a logical start, as packets filling into the same buffer, should have closer interactions to an extent. We justify our choice of 48 as a final sequence length, as given our infrastructure, the Transformer performed reasonably well on these numbers in reasonable training time.[5] Thus, we show that this aggregation is beneficial for both efficiency and performance, but it remains an open question which levels of aggregation generalize best.[6]

[3]This is similar to the pixel patch aggregation in ViT[19].
[4]We do explore the results of fixed aggregation sizes in Chapter 4.
[5]We will discuss this in further detail in Chapter 4.
[6]We will revisit this in the Chapter 5.

3.2.4 Training objectives

We aim to learn fundamental network dynamics, for which we need a pre-training task which allows us to learn this information from the packet data. As mentioned in Section 3.2.2, we use end-to-end delay as the feature which represents the fate of the packet in the network. As we argued, almost everything in a network affects packet delays (e.g. path length, buffer sizes, etc.). This makes prediction of delays a natural first choice for learning network dynamics, and we need to design a method which allows us to learn this representation. We also know that we can only generalize to a large set of fine-tuning tasks if we have a generic and robust pre-training process. Keeping all this in mind, and given the nature of end-to-end delays as we mention, we feel that capturing the dynamics using this, should be a generic enough task to learn the underlying dynamics.

We take inspiration from masked language models[53] in NLP used for training, and argue that masking certain values in a sequence, and reconstructing them, allows for effectively learning the structure in the sequence.

For our initial experiments, we mask the delay value of the most last packet in every input sequence to pre-train our NTT, as shown in Figure 3.5. We used as a fixed mask, *the number* 0, to replace our actual value, exploiting the fact that the value 0 never appears in our training data in either the delay or the size values. We can also extend this mask value to more complex data with more features, combining our domain knowledge in networks and shifting possible actual 0 values by a given constant, using the argument that shifting all values by a constant doesn't change the structure in the data. After the packet is masked, the input sequence is passed through the embedding layer, the transformer encoder, and finally we use a multi-layer perceptron as a decoder to predict the actual delay. As in this pre-training method, we do not have to worry about any of the packets which are aggregated, as those packets are never masked.

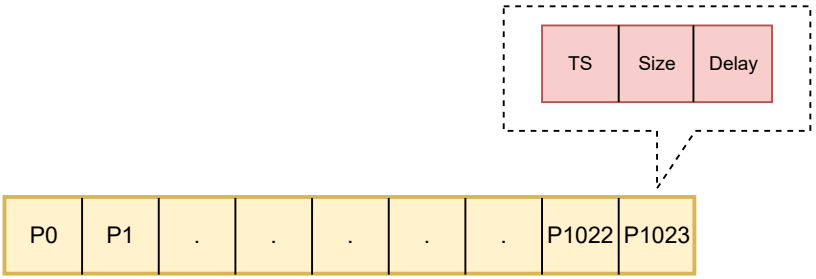


Figure 3.5: P0-P1023 is our sequence of 1024 packets each of which have 3 features namely timestamp (TS), size and delay. We mask the delay in the last packet in every sequence.

During pre-training, the NTT must learn which useful representation from the input features using the embedding layer, how to aggregate them over time using the aggregation layers, and how the aggregated elements influence each other using the transformer encoder layers. Following this, during the fine-tuning, we replace replace the decoder(Figure 3.4) to adapt NTT to a new environment(e.g. a different network) or to new tasks(e.g. predicting message completion times).This is efficient as the knowledge accumulated by NTT during pre-training generalizes well to the new task.[7]

Following the initial experiments, we also extend our pre-training process to include the following, newer techniques, with the hypothesis that this makes the pre-training process harder and in turn, more generic. We expect this to be beneficial as it prevents the NTT from overfitting on using

[7]We demonstrate the findings in Chapter 4.

all elements in the sequence to always predict a value at the same position, which in our initial design was always at the last position. In order to achieve this, we devise multiple new pre-training methods, where instead of always masking the final delay position, we mask different positions in the sequence, and try to reconstruct the masked value as the pre-training objective. Inspired by the BERT[16] model, we expect to incorporate better bi-directionality in learning sequence structure, given that the position of the masked value to be predicted is not at the end of the sequence, and has neighbouring elements on either side. This in turn, should allow for better learning of sequential structure and dynamics from the packet data.

In our initial NTT design, we only masked the last delay value which was never aggregated and the NTT could directly be trained by reconstruction and comparison against the true value. Now that other delay values will be masked, some of which might be aggregated as per our NTT's design, we propose the modified training objective.

- *For non-aggregated values:* If the masked delay value in our NTT's input sequence is not aggregated, it corresponds to a single encoded state after passing through Transformer layers. In this event, we train the model by computing the loss against the true delay value, in the given input sequence which was masked out.
- *For aggregated values:* If we mask a single delay value in our NTT's input sequence, which is then aggregated, there no longer exists a single encoded state, corresponding to this exact value. To handle this, we mask all delay values which are aggregated to the same encoded sequence, in case the individual value is in the aggregated portion of the input sequence. This means that if the delay value is in the twice aggregated subsequence, we mask out 512 values at a time and if it is in the single aggregated subsequence, we mask out 32 values at a time. To train the model now, we compute the loss against the mean delay of all actual values, which were masked out in this instance.

The *key idea* here is that packets further in the past are hypothesised to have a smaller effect on the value of the current delay, and it should hence be enough to use the average values of delays to train the model to learn the representation from these delays. A-priori, it is not trivial to decide which is the best way to mask variable positions in the input sequence, in order to have better learning of the underlying sequence structure and dynamics. In the scope of our current work, we experiment with several different variable masking strategies, in order to demonstrate that there is not (yet) one true method, which solves this problem universally.[8]

[8]We present a detailed evaluation on the performance of these NTT pre-training variants in Chapter 4.

Chapter 4

Evaluation

In this chapter, we present the first set of results and insights drawn from our NTT architecture. We begin with an overview of the data and setup for evaluation, followed by an initial evaluation and use the results from this, to draw further insights and carry out further, more robust evaluation.

4.1 Initial Evaluation Setup

We have already discussed in detail the dataset used for pre-training in Section 3.1 and the objectives in Section 3.2.4. For testing the initial performance of our pre-training process, we reserve a random 10% of the sequences from our pre-training data. This data is always used just for testing, the model does not see this data during the pre-training phase ever, and the testing dataset is always the same across any kind of comparing experiment.

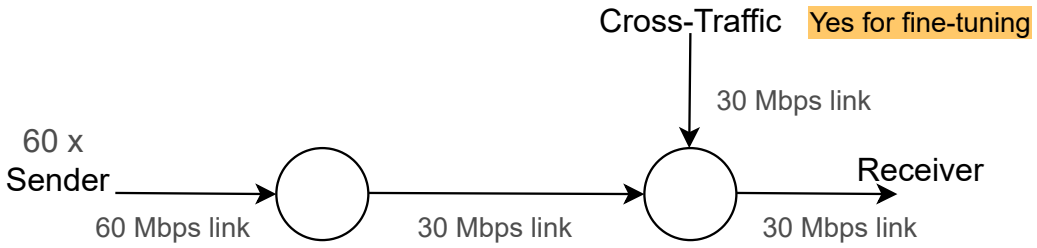


Figure 4.1: Fine-tuning data generation

The setup for the fine-tuning dataset is the same, except that we add a second bottleneck for sender traffic by introducing 20 Mbps of TCP cross-traffic on the rightmost link. The dataset does *not* explicitly contain the cross-traffic; we *only* collect packets from the senders. However, the introduction of this cross traffic affects the dynamics of the overall network due to congestion, which in turn affects the dynamics in the behaviour of the applications sending traffic. We demonstrate this setup in Figure 4.1. For our initial evaluation, we generate one simulation run of fine-tuning data only and this dataset contains about 115 thousand packets, i.e. roughly $1/10^{\text{th}}$ of the pre-training dataset. To ensure our training is not affected by extremely large possible ranges of values, we normalise our input features of size and delay, over the mean and standard deviation of each feature over the entire data, individually.[24] Based on the fine-tuning dataset generated, we have two fine-tuning tasks

- *Fine-tuning on new environment:* In this fine-tuning task, we try to predict the last delay in the sequence, on data generated on a new topology with 2 bottlenecks, as opposed to pre-training where we had 1 bottleneck. Our pre-training process employs multiple kinds of masking over different delay positions, but for the fine-tuning task objective is prediction of the delay at the last position. The *key idea* is to use the learn sequence structure in the during the pre-training and apply it to predict the last delay and make a decision on the final packet during the fine-tuning, following a change in network environment. We expect that if the pre-training is done robustly, fine-tuning in this new environment on a similar task will be both faster and require significantly less data.

- *Fine-tuning on a new task:* In this fine-tuning task, we try to predict the Message Completion Time (MCT) of an entire transmitted message, given the NTT's encoded output states of the previous 1024 packets and the size of the given message. Our fine-tuning objective is to mean pool[61, 63] over all the NTT's output states which represents the overall learnt behaviour of the network dynamics in the recent past. We combine this learnt feature with the message size, which we can have in our simulation data using appropriate features from our Network Simulator[47] and use this to predict the MCT using a Multilayer Perceptron with linear layers. The *key idea* is to learn the *overall* structure from the network dynamics and use it to predict tasks, which depend on the nature of these dynamics.

For evalaution, we calculate the mean-squared error (MSE) for achieved on both tasks and report our initial findings in Table 4.1. We process MCTs on a logarithmic scale to limit the impact of outliers. This is required as our MCT dataset is quite heavy tailed, which hurts training deep learning models.[1]

4.2 Preliminary Results

In our initial experiments, we compare several versions of our NTT, vary both our pre-training and fine-tuning objectives, in order to explore the possibilities of what information can actually be learnt and to which extent. We focus on evaluating and demonstrating on the following points:

- Our NTT is capable of learning network dynamics from packet data
- Pre-training enables our NTT to generalize on new environments and new tasks
- Incorporating our networking domain specific knowledge helps the NTT perform better

We demonstrate our initial comparison in Table 4.1 The *pre-trained* model first learns from the pre-training and then from the fine-tuning dataset, while the *from scratch* version only learns from the fine-tuning dataset. In these experiments, the fine-tuning dataset is only around $1/10^{\text{th}}$ of the size of the pre-training dataset. In all the NTT variants presented here, we only mask the last packet delay during the pre-training process.[2]

In order to evaluate the choice of our feature selection for pre-training , we perform some ablation studies on the input data. We argue that we need both packet state information and network state information to effectively learn network dynamics. In order to investigate this, we pre-train two additional versions of the NTT, one *without delay* and one *without packet size* information, in the respective input sequences. For both our pre-training and fine-tuning, we reserve a part of the

[1] Our simulation's mean and 99.9th percentile MCTs are 0.2 and 23 seconds, respectively.
[2] Further details on training (e.g. hyperparameters) have been moved to the Appendix A.

datasets for testing, which is always constant, unknown to the model and same across every NTT variant.

We also perform ablation studies to investigate the effects of aggregating features in our input, which we feed into the encoder stage of our NTT. Concretely, we compare our multi-step aggregation of 1024 packets into 48 aggregates (see Section 3.2.3) with *no aggregation* (using only 48 individual packets) and *fixed aggregation* (using 48 aggregates of 21 packets each, i.e. 1008 packet sequences). Due to limits of our training infrastructure and Transformer's quadratic scaling behaviour with increasing sequence size, it is infeasible to evaluate the effects of having no aggregation on very large sequences.

all values $\times 10^{-3}$	Pre-training	Fine-tuning	
	Delay	Delay	log MCT
NTT			
Pre-trained	0.072	0.097	65
From scratch	-	0.313	117
Baselines			
ARMA	1.800	1.180	1412
Last observed	0.142	0.121	2189
EWMA	0.259	0.211	1147
NTT (Ablated)			
No aggregation	0.258	0.430	61
Fixed aggregation	0.055	0.134	115
Without packet size	0.001	8.688	94
Without delay	15.797	10.898	802

Table 4.1: Mean Squared Error for all initial NTT models and tasks. The MSE is computed on the task specific actual value vs the predicted value of the particular model. The MSE is always calculated on the same test dataset.

We also to ensure that we understand if our NTT learns useful information in the first place. To put this into perspective, we use the following three simple baselines in order to compare the performance of our different NTT variants.

- *Auto-Regressive Moving Average (ARMA)[58]:* In this method, we predict the mean delay of all the previous delay values in our sequence as our target delay value.
- *Last observed:* In this method, we predict the last observed delay over all the previous delay values in our sequence as our target delay value.
- *Exponentially Weighted Moving Average (EWMA)[10]:* In this method, we predict a weighted smoothed mean delay of all the previous delay values in our sequence as our target delay value.[3]

NTT can learn some network dynamics: Based on the initial evaluation, we confirm that the *pre-trained* NTT beats our baselines in all cases. Though the baselines are rather basic, we can already conclude that the NTT learns the network dynamics to an extent and that the the error

[3]For our EWMA, we use the decay factor $\alpha = 0.01$

values are sensible. We also observe that some of the baselines work reasonably, depending on the complexity of the task. For e.g. the smoothed *EWMA* approach works reasonably well for delay prediction, it fails for the more complex MCT task. On the delay task, we further observe that the *last observed* baseline performs better than *EWMA*; that is, smoothing over a long sequence performs worse than simply returning the most recent packet. This suggests that there is something to learn from the *sequence dynamics*, which the NTT can do much better, as we see it perform better for both types of fine-tuning tasks. This suggests that the NTT can learn fine-grained sequence information, along with aggregated structural information from the entire sequence.

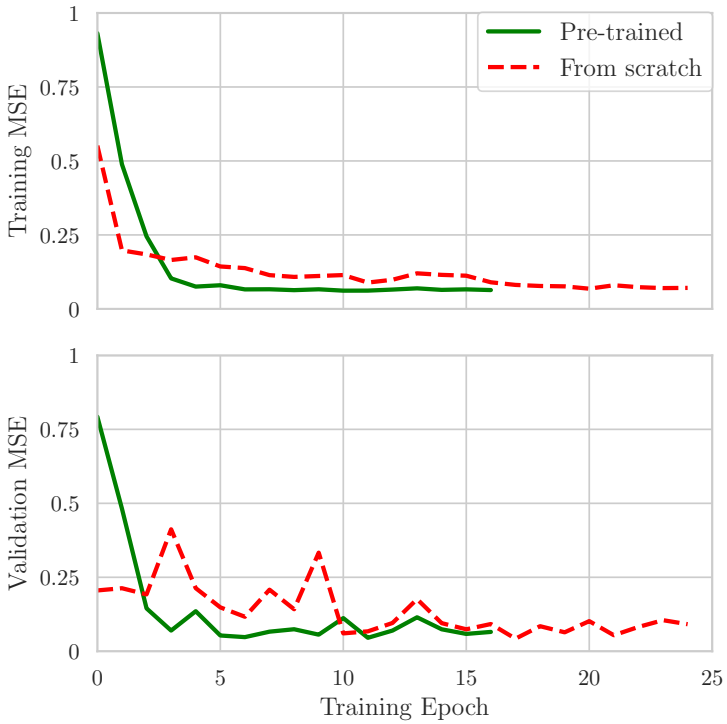


Figure 4.2: Mean-square error for prediction of message time completion over training epochs

NTT can generalize across environments and tasks: We observe that pre-training is beneficial: on both fine-tuning tasks, the *pre-trained* NTT outperforms the *from scratch* version.The pre-training knowledge generalizes to a new environment i.e. unobserved cross-traffic and to a new task i.e. MCT prediction. In our current NTT variants, we pre-train and fine-tuning for delay prediction in the exact same way, by learning to predict the last delay in the sequence. Pre-training also improves the speed of the fine-tuning process, in addition to the overall performance. We see this in Figure 4.2 shows that pre-training allows faster learning on the fine-tuning dataset: the *pre-trained* NTT reaches a lower error after a few (only 3) training epochs and finishes training, i.e. no further improvement, after fewer epochs (17 vs 25), with a lower MSE on the test set in the end (Table 4.1)

NTT specific features help: Finally, we argue about about the benefits of the two NTT-specific features we propose: i.e. the hierarchical aggregation layer and the mix of network and traffic information in the raw data.

With *no aggregation*, the model has little history available; we observe that, perhaps surprisingly, this affects the predictions on the delay, but not on the MCT. Practically, our NTT will not scale

to arbitrarily large sequences, so trying to learn from longer sequences with no aggregation is irrelevant. Conversely, with a *fixed aggregation*, the model loses recent packet-level details but has access to longer history; this seems sufficient to predict delays but affects the MCT.[4] It is a little early to map the exactness of these effects at this point, however we can conclude from this initial result suggests that both recent packet level information and longer historical aggregates are useful to generalize to a large set of tasks.

When we consider the NTT versions *without packet size* and *without delay* information (Table 4.1), we observe that neither generalizes well. Without the packet size, the model overfits the pre-training dataset and performs poorly on predicting delay for fine-tuning. This is not surprising as given only a single feature, the model cannot learn relational information well. Without delay information, the model does not produce any useful prediction related to packet delays or MCTs. This is also not surprising as we cannot expect to predict any useful information about the network dynamics, without using any data from it in the pre-training process.

Based on the promising nature of our initial results, we now move to more robust evaluation, harder and more generic pre-training and fine-tuning objectives, in order to explore to what further extent can our NTT perform. We explore further methods to analyse and improve our NTT in the following Section 4.3.

4.3 Further Results

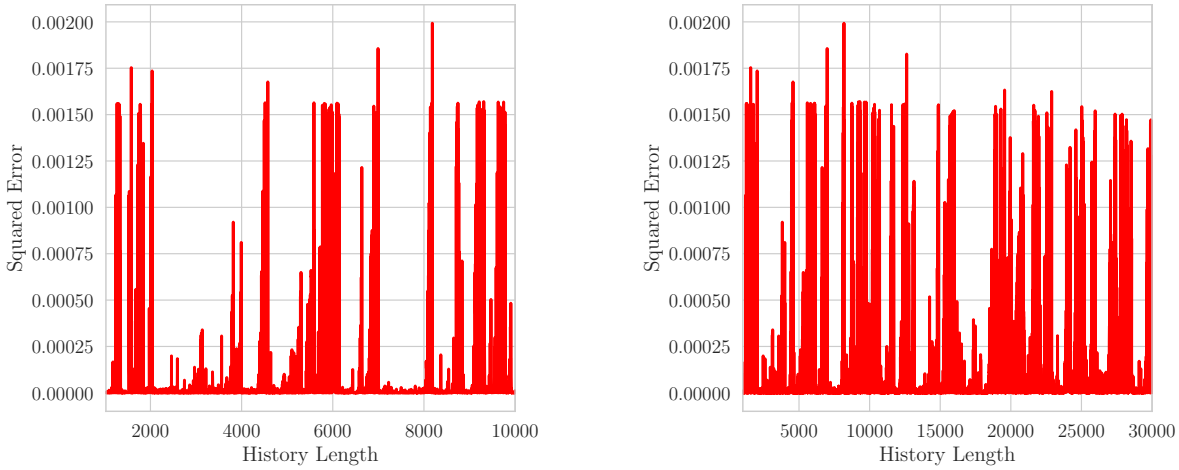
4.3.1 NTT vs improved baselines

Our initial experiments show that our NTT learns sequence structure from the packet data reasonably well, and we have several baselines to argue that we learn useful information. Since our initial baselines are rather simple, we now present further analysis against more improved and tuned-baselines, in order to understand the learning behaviour of our NTT better. Our objective of using these new baselines is *NOT* to show that our NTT outperforms them by significant order (given that we work in a small setup with limited data), but rather to put into perspective the nature and complexity of learning the sequence structure of our in general. We argue that learning structure from our data is not trivial, even across a variety of models which learn structural information in very different ways.

Auto Regressive Integrated Moving Average: We first present an Auto Regressive Integrated Moving Average (ARIMA)[9] model as a method to learn structure in the data. As the ARIMA algorithm is capable of only learning from a single time-series of real numbers, we present it the past history of *delay values*, and fit the model on these values, in order to predict the subsequent delay value. Our choice of this model as a baseline is due to the fact that the ARIMA model has successfully been used in several types of time-series forecasting[54]. We have a *one-step rolling* ARIMA forecast, where at every stage, we fit the entire history of all previous delay values to predict the next delay, following which we shift our window forward by 1, and carry out the same procedure. We carry out the ARIMA rolling forecast over the delay values from our pre-training dataset (see Section 3.1) and keep the minimum past history size of 1023 values, in order to have enough past values and also in conjunction with our bottleneck queue size, as we did for the NTT's sequence definition.

[4] In the vision transformer[19], they got extremely good results with fixed size aggregations and very large training datasets.

We refer to the learnt behaviour by fitting ARIMA in Figure 4.3 and present two models, one with increasing history size up to the previous 10000 and another with upto 30000 past delay values. It is evident that the squared errors in the ARIMA models do not stabilise even after fitting on increasing past history sizes. This shows that the structure in our packet data cannot be trivially learnt by just fitting on previous values, we need some features which help understand the network dynamics, and this is the very problem we try to solve using our NTT. Even in terms of efficiency, the one step rolling forecast ARIMA model on increasing history size, doesn't scale. We require a fitting time of ∼1.5 hours for a past history window size upto 10000 and ∼13 hours for a past history window size of 30000, which also practically limits us from fitting much larger windows. In similar time orders to fitting on upto the 30000 past history, our NTT learns network dynamics to a significant extent and is able to generalize to new tasks (demonstrated in Table 4.1).



(a) ARIMA on history upto past 10000

(b) ARIMA on history upto past 30000

Figure 4.3: Squared error on the predictions of the ARIMA models trained to predict on a one-step rolling forecast

Bi-Directional Long Short-Term Memory: Long Short-Term Memory (LSTM) memory networks[28] have been the state-of-the-art models for sequence modelling for many years, before the introduction of the Transformer architecture. Prior to the same, LSTMs have successfully been used to successful solve several sequence-to-sequence tasks in NLP and time-series forecasting. Being a specialised RNN architecture, LSTMs suffer from the weaknesses of dependence on sequential processing and also vanishing gradients compared to the Transformer, as discussed in Section 2.1.1. However, in our project, it is is useful to include it as a baseline comparison, as it is a deep learning model designed to solve a very similar task. This in turn is helpful for us, in order to understand the learning behaviour of the Transformer to a greater extent.

We choose a Bi-Directional LSTM (Bi-LSTM)[14] architecture as a baseline, as it was the best known LSTM model for sequence tasks, prior to the use of Transformers becoming the norm for similar problems and present several variants of a trained Bi-LSTM model. In order to keep the comparison meaningful, we ensure that the total number of trainable parameters in the Bi-LSTM models, are similar to those in our NTT (∼$4M$ parameters). We only compare the pre-training task using the Bi-LSTM model, for our baseline. The pre-training process is the same as the NTT, we mask the *last delay* in the in the input sequence of features, in the exact same manner in which we pre-train our initial NTT (as discussed in Section 3.2.4). The pre-training dataset is kept constant

across all these models for uniformity in learning.

all values $\times 10^{-3}$	Pre-training by masking last delay
NTT	
Standard	0.072
Bi-LSTM	
Aggregation + input embedding	0.062
Aggregation + no input embedding	15.8
No aggregation + no input embedding	-
No aggregation + input embedding	-

Table 4.2: Mean Squared Error on pre-training NTT vs Bi-LSTM. The MSE is calculated on the predicted delay vs the actual delay value.

The evaluation of pre-training Bi-LSTMs has been summarised in the Table 4.2. A-priori, we do not assume that the Bi-LSTM requires input data in a form equivalent to the NTT, in terms of aggregating our input features and passing the input features through an embedding layer before feeding it to the Bi-LSTM encoder layers. However, it is observed that the Bi-LSTM model learns efficiently and equally well as our NTT, only when we provide it the input data in the exact same method. The fact that it learns well under these conditions is not surprising at all, given that our aggregated sequences are not very long (limited to 48 as of now). However the pre-training time for our NTT is much shorter ($\sim$12 hours till convergence) as compared to pre-training the Bi-LSTM with the same input ($\sim$19 hours till convergence) on identical hardware resources. We observe that if we do not use an embedding layer on the input to the Bi-LSTM, it performs poorly on the same pre-training objective, re-emphasising that such a learnable embedding is necessary to learn useful information from the features. Training the Bi-LSTM on non-aggregated input features (sequence size of 1024) doesn't converge in practical time periods (a single epoch takes $\sim$21 hours to run on the same hardware) and hence, we do not evaluate those experiments further in our current scope.

4.3.2 Evaluation on robust fine-tuning

In the process of fine-tuning on our initial pre-trained NTT architecture, we showed that it generalised to predicting the *last delay* value in a new environment, where the topology introduces 2 bottlenecks using cross traffic (see Figure 4.1). The fine-tuning data was much less than the pre-training data, and we argued that with generic enough pre-training, we could achieve extremely good performance on new datasets with relatively less amounts of fine-tuning. We now analyse this behaviour further and demonstrate the pre-training on a single bottleneck indeed helps in generalizing to topologies with multiple bottlenecks, in terms of requiring less data or less training effort or both. For these experiments, the pre-training process involves masking the delay value at the *last position*.

For this, we generate a large amount of fine-tuning data, which is comparable in magnitude to the pre-training data ($\sim$1.2 million packets). There is always, as a testing dataset, a constant subset of the fine-tuning dataset, which is unknown to and identical for every model variant. The NTT is now fine-tuned in multiple ways, we use the pre-trained NTT and fine-tune it on the entire fine-tuning dataset and separately on only 10% of the fine-tuning dataset. Simultaneously, we also fine-tune the NTT with randomly initialised weights, both on the entire fine-tuning dataset and on only 10% of the dataset. The 10% subset of the fine-tuning data is the same across all variants.

Model	MSE: Delay Prediction *all values*$\times 10^{-3}$	Layers trained	# of Epochs trained
NTT			
Pre-trained + Fine-tune (full)	0.033	MLP Head only	5
Pre-trained + Fine-tune (10%)	0.037	MLP Head only	9
From scratch + Fine-tune (full)	0.036	Full NTT	13
From scratch + Fine-tune (10%)	0.118	Full NTT	15
Pre-trained + No Fine-tune	0.108	-	-

Table 4.3: Comparing NTT across fine-tuning variants. The MSE is calculated on the predicted delay vs the actual delay value.

The results on the fine-tuning variants are summarised in Table 4.3. From these experiments, it is evident that the initialising the NTT with *pre-trained* weights helps the fine-tuning. With pre-training, using the entire fine-tuning dataset, we can reach a low MSE value in very few fine-tuning epochs (only 5), whereas when we use only 10% of the fine-tuning data, we can still reach similar values of MSE with just a few more epochs (only 9). Also, with pre-training, it is only required to train the linear layers at the end of the NTT, which also reduces the training time on the whole. When we initialise *from scratch*, we need much more effort to reach similar performance on the fine-tuning dataset. Using the entire fine-tuning dataset in this case, we achieve the same order of MSE, but only with training the entire NTT, for many more epochs. This behaviour is as expected as it is equivalent to performing pre-training on the fine-tuning dataset of approximately equal size. However, if we use only 10% of the fine-tuning data in this case, we hardly learn anything which is seen in the MSE, as in this case, it is the same level of performance as initialising with pre-trained weights and not performing any fine-tuning. Based on these, we can effectively demonstrate that pre-training our NTT, does indeed help in a much quicker fine-tuning process i.e. the NTT is able to generalise on learnt behaviour during pre-training, on tasks similar to the pre-training.

4.3.3 Improved pre-training for NTT

We also evaluate our NTT by pre-training it with different masking strategies. Concretely we no longer mask only the last packet delay in the sequence for pre-training and learning the sequence structure, but different positions in the sequence, as introduced in Section 3.2.4. The challenges and approaches to combining the masking of delays in the past, and the aggregation levels have already been discussed there; we now evaluate and compare the performance across these pre-training variants. The fine-tuning objectives haven't changed, we still use the same two fine-tuning environments, *first* where we try to predict the delay of the last packet in the sequence data generated from the 2 bottleneck topology and *second* where we try to predict the MCT, based on the most recent packet data. Given the nature of the new pre-training, our primary expectation is that the MCT prediction task should benefit from it. This is because for MCT prediction, we use the average pooled encoded state from the NTT's Transformer Encoder output stage and we expect the information contained in this "average state" to be higher with better, overall learning sequence structure, enabled by bi-directional pre-training.

The final challenge is to decide on the frequency of masking packet delays across variable levels of packet aggregation i.e. how often should one mask delays from non-aggregated packets and how often should one mask delays from aggregated packets. To handle this, we experiment with several masking strategies, in increasing order of complexity and number of elements masked at every step,

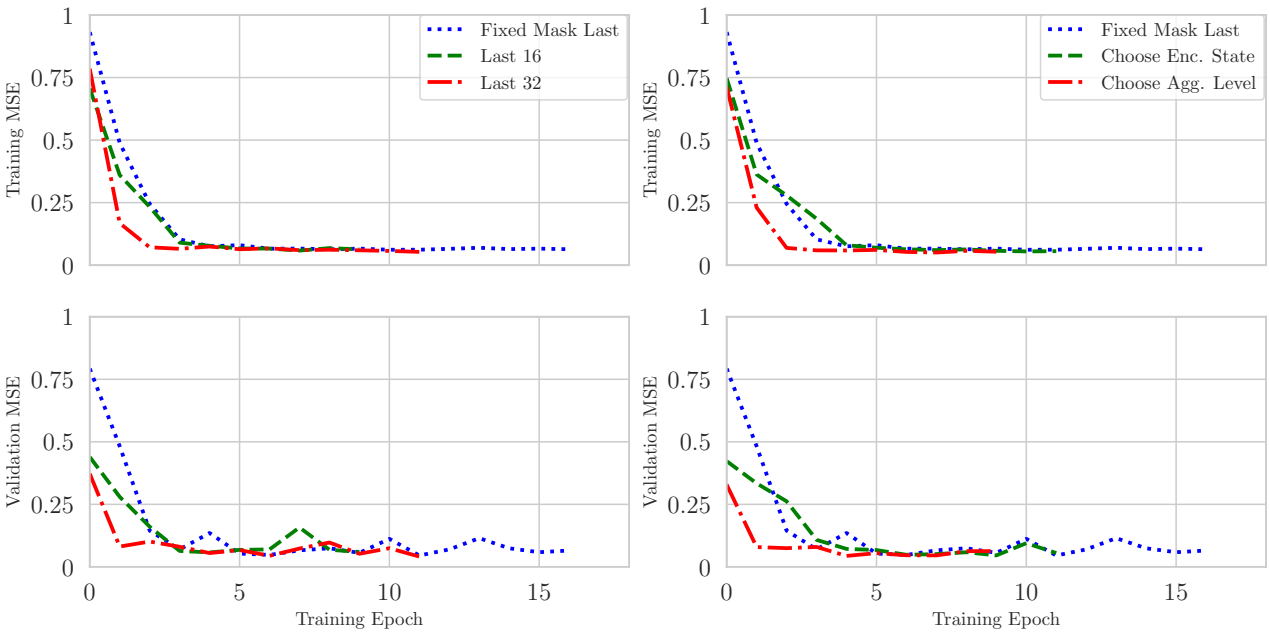
and evaluate the performance in every case.

- *Choose from last* 16*:* We always mask one packet delay, by choosing uniformly over one element in the 16 most recent packets. In this method, we never mask a delay from a packet which is aggregated.
- *Choose from last* 32*:* We always mask one packet delay, by choosing uniformly over one element in the 32 most recent packets. In this method, we also never mask a delay from a packet which is aggregated.
- *Choose from NTT encoded states:* We choose uniformly over all 48 encoded states of the NTT's output. If the encoded state corresponds to a single non-aggregated packet, we mask that packet's delay value. If the encoded state corresponds to aggregated packets, we mask all packet delays aggregated to that given value. In this method, we do not distinguish between levels of aggregation.
- *Choose from aggregation levels:* We choose uniformly over all 3 (non)aggregation levels. If we end up with a non-aggregated level, we mask 1 delay uniformly chosen over the non-aggregated packet delays. If we end up with *one level* of aggregation, we uniformly choose 1 aggregated value and we mask all packet delays aggregated to the chosen encoded state. If we end up with *two levels* of aggregation, we mask all the packet delays in the first half (512) of the sequence, which aggregate to the chosen encoded state.

all values $\times 10^{-3}$	Pre-training	Fine-tuning		
	Delay	Delay(on full)	Delay(on 10%)	log MCT
NTT: Chosen mask				
From last 16	0.054	0.042	0.046	64
From last 32	0.063	0.052	0.057	55
From encoded states	0.063	0.049	0.053	51
From aggregation levels	0.087	0.053	0.057	61

Table 4.4: Mean Squared Error for all differently masked pre-trained NTT models and tasks. The MSE is calculated on the predicted vs the actual value.

Referring to Table 4.4, we see that the different masking strategies do help learning structural information from the sequence of packet data better, especially for the MCT prediction task. If we compare with Table 4.3, we see a slight drop in performance when we fine-tune on the last position delay prediction, as opposed to the time when we pre-trained the NTT to predict the last delay too. This is expected behaviour as now the pre-training and fine-tuning tasks for delay prediction are not identical. If we compare with Table 4.1, we do see quite a bit of improvement in the MCT prediction. We evaluate the MCT prediction further and refer to Figure 4.4. We see that with variable position masking, we reach lower levels of error with much fewer epochs of training in general, as opposed to when we have the masked pre-training only on the last delay position, which is enabled by the bi-directionality in the masked learning process. Given our current datasets, it is not possible to conclusively decide on "one best" masking strategy for pre-training. However, we can definitively conclude that masking on variable position does help in better network dynamics learning from the sequence and in turn, better generalization, which is our ultimate goal.



(a) Pre-train with no mask over aggregated delays (b) Pre-train with also mask over aggregated delays

Figure 4.4: Mean-square error for prediction of message time completion over different pre-training masks

Finally, the NTT architecture for pre-training needs to be modified slightly, when we change the masking strategy for learning. For the cases of *choosing from last* 16, *choosing from last* 32 or *choosing from NTT encoded states*, we use the standard NTT model, with aggregation, followed by a Transformer encoder, followed by a set of linear layers. This method works well in all of the above cases, but does not work in the case of *choosing from aggregation levels*. The most probable reason for this is, that due to equally choosing between the levels of aggregation, every $1/3^{rd}$ of the choices, a completely different type and number of delay values are masked. To handle this, we use three independent instances of the same set of linear layers, one for each kind of aggregation level, in order to effectively learn from each aggregation, and not have a single set of neurons "pulled" in different directions for every type of masked value.[5]

4.3.4 Evaluation on multi-path topologies

Till now, our experimental studies have been carried out on the single link topology as shown in Figure 4.1. Our feature from the network state to learn the network dynamics is *end-to-end delay*, which is the combination of the *queuing delay* and the *path delay*. As we had a single link topology for our evaluation, the *path delay* was always constant and learning the dynamics was dominated by the queuing delay factor. We move to evaluation on a larger topology now, with different paths of multiple lengths, in order to also have more contribution from the path delay in our *end-to-end delay* metric. Our method of traffic generation is similar to before, we have multiple applications sending traffic but we also have multiple receivers over different paths. For evaluation, we generate 10 runs of fine-tuning data, which corresponds to ∼1.7 million packets, following which we use the sequences generated from this data, to fine-tune on our NTT model. We have 20 Mbps of cross

[5]We move further studies on using multiple instances of linear layers to Appendix B.

traffic on the on each of the cross-traffic sources, creating altogether 4 bottlenecks in the network setup, as shown in Figure 4.5.

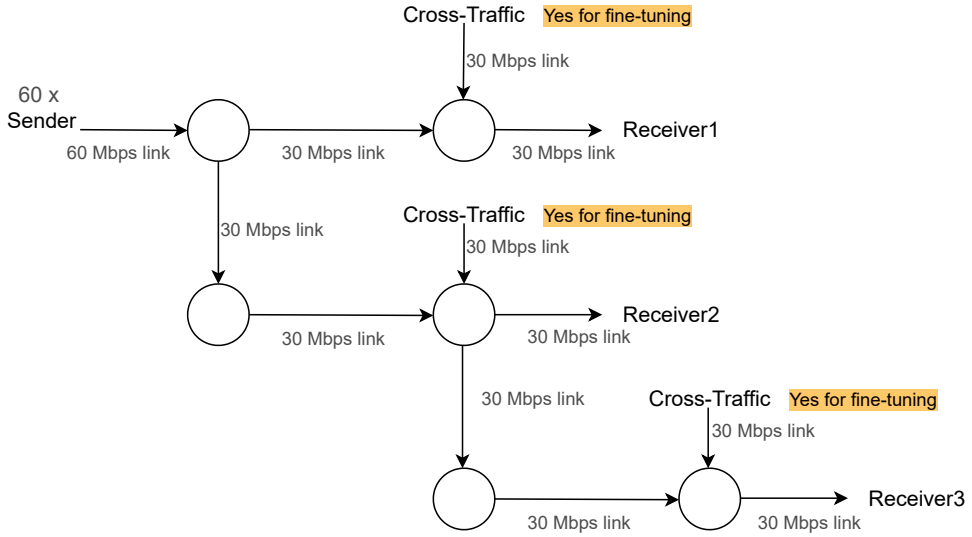
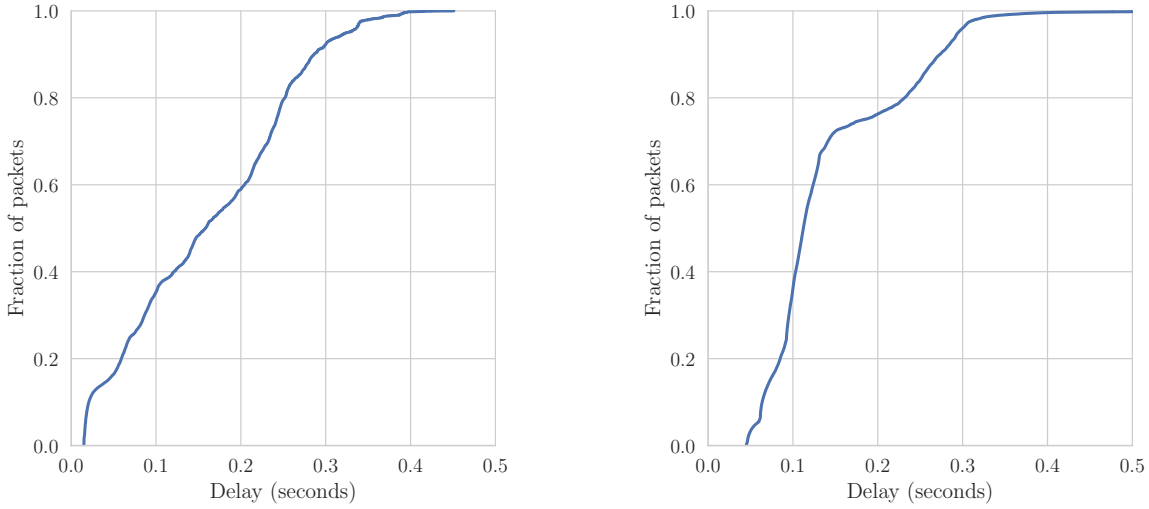


Figure 4.5: Fine-tuning data generation on larger topology

As we now fine-tune on topologies with multiple paths and more complex traffic patterns, it is necessary to ensure that the distribution of the overall traffic (which represents the underlying dynamics) actually changes with the new topology. For this, we refer to Figure 4.6 and compare the delay distributions between the topology in Figure 4.1 (on the left) and the topology in Figure 4.5 (on the right), to ensure that the overall behaviour of the data to be learnt, actually changes.[6]



(a) Delay CDF, 2 bottlenecks, single run

(b) Delay CDF, 4 bottlenecks, single run

Figure 4.6: Comparing delay distributions across different fine-tuning topologies

Learning the network dynamics effectively from a single bottleneck, should be enough to gen-

[6]We move more details on individual path distributions to Appendix C.

eralise to similar dynamics on multiple bottlenecks , to a reasonable extent. Hence we will use our pre-trained NTT and fine-tune the model on the packet data from the new topology. For these experiments, we will use the base NTT, which is pre-trained on masking the *last delay*. To effectively learn dynamics of packets to different receivers, we will need a feature to distinguish receivers, and for this we add a new feature to our input packet data, an integer to represent the *Receiver ID*. We train the entire NTT architecture i.e. all the layers, given the increased complexity in the new data.

Model	MSE: Delay Prediction *all values*$\times 10^{-3}$	# of Epochs trained
NTT		
Pre-trained + Fine-tune (full)	0.004	5
Pre-trained + Fine-tune (10%)	0.035	12
From scratch + Fine-tune (full)	5.2	10
From scratch + Fine-tune (10%)	8.2	15
Baselines		
ARMA	4.2	-
Last observed	11.2	-
EWMA	4.0	-
NTT (Ablated)		
Pre-trained + Fine-tune (full) : No receiver ID	2.8	8
From scratch + Fine-tune (full) : No receiver ID	2.7	15

Table 4.5: Fine-tuning NTT on larger topologies with multi-path and bottlenecks. The MSE is calculated on the predicted delay vs the actual delay value.

We refer to the results in Table 4.5 and make the following *key conclusions*, on fine-tuning and generalization behaviour of the NTT architecture.

Pre-trained NTT generalizes much better: We observe that pre-training the NTT, prior to fine-tuning really shows very good generalization on learnt bottleneck and congestion behaviour. The learning of sequences structure on this new data is much harder than before, as the same baselines of *ARMA*, *last observed* and *EWMA* perform extremely poorly, indicating the complexity increase in the data and reinforcing that underlying network dynamics need to be learnt in order to make general predictions on delay. We see that the NTT performs the best when when it is pre-trained and fine-tuned on the entire fine-tuning dataset, and also quite well when fine-tuned on 10% of the dataset. Without the pre-training, the NTT fails to learn these dynamics and performs poorly on fine-tuning, even with a lot of fine-tuning data. In our experiments, all models are trained till there is no further improvement in performance, and we see that even with many more training epochs, the NTT *from scratch* cannot learn the dynamics here. This shows that given the same model size, it is much harder to learn dynamics from complex network interactions, without pre-training on simpler data first.

Receiver ID is needed to learn dynamics: It is clear from the *ablated NTT*, when we do not use the *Receiver ID* as a feature, the NTT fails to learn in both cases. With the removal of this feature, the NTT can no longer distinguish between traffic directed to different receivers and is unable to separate the bottleneck dynamics, between different sources. In the preliminary NTT

training, this was not a problem as the NTT learnt that all congestion is always on the same link, which is not the case here. While we include this as a feature now, we cannot be certain this is the most optimal way, and future versions of NTT, may require different path identifiers. This is beyond the scope of the current project and we move this to future discussion in Section 5.2.

Chapter 5

Outlook

The evaluation for our current NTT design is extremely promising, and indicates that such an architecture can be built for learning and generalizing on network dynamics. However, the process does not end here, there is huge scope of possible future research and there exist multiple directions in which the NTT can be improved. We present some ideas which we feel can be the next steps for these improvements.

5.1 Learning on bigger topologies

We evaluate our NTT on simple topologies in our project, as we are still in the initial phase of building such an architecture. Real networks are undeniably much more complex than our current training environment. Topologies from these networks (e.g. a large datacenter, ISP etc.) are larger with traffic flowing along multiple paths; there are many different applications, transport protocols, queueing disciplines, etc. and the interactions across these traces, lead to extremely complex network dynamics. Additionally, there are many more fine-tuning tasks to consider, e.g. flow classification for security or anomaly detection. We evaluate pre-training on a small topology and its generalising behaviour on a larger topology to an extent, based on learnt behaviour of network dynamics of bottleneck queues in Section 4.3.4. However, this merely scratches the surface and doesn't match the scale of complexity of dynamics on real networks.

Apart from this, our current network traces for training and evaluation are only drawn from a small subset of Internet traffic distributions[40]. Testing our NTT prototype in real, diverse environments and with multiple fine-tuning tasks will provide many more invaluable insights. We can not only better understand the strengths and weaknesses of our NTT architecture but also into the "fundamental learnability" of network dynamics. A first step for this can be conducting more extensive, more complex simulations and analysing real-world datasets such as Caida[12], M-LAB[37], or Crawdad[15] or Rocketfuel[49]. This kind of evaluation will provide much deeper insights into the generalization on learnt network dynamics.

How does the NTT hold up with more diverse environments and fine-tuning tasks? Which aspects of network dynamics are easy to generalize to, and which are difficult?

5.2 Learning more complex features

More diverse environments, i.e. with more diverse network dynamics, also present the opportunity to improve our NTT architecture. The better the learning of general network dynamics during

pre-training, the more can other models benefit from the NTT during fine-tuning. The directions for improvement we see here are:

- *Better sequence aggregation:* We base our current NTT's aggregation levels on the number of in-flight packets, i.e. whether packets in the sequence may share their fate, determined by buffer sizes in our experiments. Evaluations show that the hypothesis holds; the further packets are apart, the less likely they do share fate. Such packets are aggregated much more, given their individual contribution to the state of current packet is much lower. Currently, we believe matching individual aggregation levels to common buffer sizes (e.g. flow and switch buffers) may be beneficial. Much more research is still needed to put this hypothesis to the test and determine the best sequence sizes and aggregation levels for the future NTT versions.

- *Multiple protocol packet data:* Till now, we did not use network traces which combined different transport protocols or contained network prioritisation of different traffic classes, and thus did not use any specific packet header information in our features for learning. Considering such header information might be essential to learn behavioural differences between such categorisations. Raw packet headers can be challenging to provide as inputs to an ML model, as they may appear in many combinations and contain values that are difficult to learn, like IP addresses[64]. Some research ideas from the network verification community on header space analysis[33] may provide valuable insights on header representations and potential first steps in this direction.

- *Learning path identification:* Currently, we do not have a concrete method for the NTT to learn differences between multiple possible paths in the network, a feature which will become significant, to learn on larger topologies. Evaluation on the topologies with multiple paths (in Section 4.3.4), demonstrated that such a distinction is indeed required. In the initial experiments, this was solved by providing a unique identifier (Receiver ID) as an additional feature in the input feature set. While this is a quick, simple fix; it might not be an optimal method to scale to larger topologies. Additionally, such a simple identifier does not provide insights about hierarchical overlap (e.g. subnets), which might be required for more efficient learning. Networks today learn the required path information from the routing function (e.g. shortest path, prefix matching, subnetting etc). While it might be possible for the NTT to "learn" the the path information and the routing function by giving it all features like prefix, subnet mask etc., this might be suboptimal, hard and unnecessary. Coming up with possible ideas to learn path information better is a possible next step we see to improve the NTT.

- *Dealing better with rare network events:* While in several fields of machine learning, it is enough to learn behaviour "on average", this doesn't translate to the domain of network data. Less frequently occurring events in networks (e.g. packet drops from link failures) can lead to significant information loss. This kind of behaviour is hard to learn for machine learning algorithms due to these events have relatively very less representation in the training data but it is essential that our NTT learns outcomes of these events to an extent. One possible step is to collect telemetry data like packet drops or buffer occupancy as features. This may allow the model to learn the behaviour of networks better, though this will be hard to due to sparse nature of such data and future research is needed to solve this in an efficient way.

How can we improve the NTT design to learn efficiently from diverse environments? How can we deal with an information mismatch between environments?

5.3 Collaborative pre-training

Transformers in NLP and CV have only shown to truly outshine their competition when pre-trained with massive amounts of data. We envision that this would require a previously unseen collaboration across the industry. We see two main challenges:

- *Training data volume:* Training an NTT to learn complex dynamics at the scale of large topologies will require a extremely large amount of data. Given the differences possible across networks, the pre-training data will need to be representative of this, which will require huge amount of network traces, which no single organisation might have access too and will require collaboration between severalof them.
- *Data privacy:* Due to privacy concerns, it might not be possible to share a lot of this data publicly. Also several organisations might be unwilling to share their data anyway, as it would cost them their competitive advantage in the industry.

We see some possible solutions to these problems. ML models are known to effectively compress data. As an example, GPT-3[11] is one of the largest Transformer models for text data to date and consists of 175 Billion parameters or roughly 350 Gigabytes. However, it contains information from over 45 Terabytes of text data. Another huge model is Data2Vec[5] which is a general purpose Transformer which learns representations for text, images and audio using a task-agnostic training approach with knowledge distillation[27], but is trained on trillions of datapoints. Sharing a pre-trained model is much more feasible than sharing all the underlying data, it also reduces training time and redundancy in re-training for already established results. Furthermore, sharing models instead of data could overcome privacy barriers via federated learning[32]. Organizations can keep their data private and only share pre-trained models, which can then be combined into a final collectively pre-trained model. This brings up the problem of how can these models be trusted, but this can be solved by making the details of the pre-training process public, while keeping the training data private.

Can we leverage pre-training and federated learning to learn from previously unavailable data?

5.4 Continual learning

Language and image structure does not evolve much over time. A cat's remains a cat's image, whether viewed today or 10 years later. A sentence in English might change slightly over time, due to changes in grammatical rules but the overall structure is similar. Pre-trained models on such data thus, do not need to be re-trained frequently. However, the Internet is an ever evolving environment. Protocols, applications, etc. may change over time. Interactions in networks due to addition of new network nodes sending traffic, may change the underlying network dynamics significantly.

Though we are certain that underlying network dynamics will change over time, we do expect them change less frequently than changes in individual environments, and still argue that the same pre-trained NTT may be used for significant time, with just small amounts of fine-tuning from time to time. Nevertheless, at some point, even the learnt NTT model on underlying dynamics may become outdated and will have to be re-trained. It is already difficult to determine when it is helpful to re-train a specific model[60], and for a model that is supposed to capture a large range of environments, this is very likely be an even harder task.

At which point should we consider an NTT outdated? When and with what data should it be re-trained?

Chapter 6

Summary

We have seen that earning fundamental dynamics in networks is not a trivial problem. Even on the simplest network topologies, different interactions of traffic arising from multiple sources, can create complex patterns in the overall traffic. However, amidst these complexities, there exist some underlying patterns, which can effectively be captured and learnt, by having the right model equipped for this task. Over a span of time, measurements taken on these traffic complexities, show that not all is truly random, there is some structure which can be learnt, and be doing so, this can be leveraged to improve performance in networks on a whole.

We present in this thesis, a new NTT model, which serves the first steps towards learning fundamental behaviour of dynamics from network traffic. Based on state-of-the-art techniques on learning sequences in data in other fields, we present our Transformer based architecture which is trained to learn similar sequential information in network traffic data. Of course, doing so even in our setup, is quite challenging and there is clear scope for a huge amount of improvement. We feel that this approach surely opens up a plethora of new research questions and directions in the field of learning for networks. Over the course of this project, we explore several possible methods of pre-training on traffic traces, following which we evaluate and compare them, in order to understand better, the possibilities and limits of learning the network dynamics. Based on our findings, we do conclude that learning these networks dynamics is definitely possible, and acknowledge the fact that a lot is still unknown about deep learning on such data, but at the same time, realise that we are in a much better position to decide new directions to proceed in, given our current NTT architecture.

We hope that our initial work in this direction, motivates the networking community to explore the vast possibilities of taking a step further in this domain and working together to build smarter and better learnt models, to improve performance and efficiency in the networks of tomorrow.

Bibliography

[1] Abbasloo, S., Yen, C.-Y., and Chao, H. J. Classic meets modern: A pragmatic learning-based congestion control for the internet. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2020), SIGCOMM '20, Association for Computing Machinery, p. 632–647.

[2] Akhtar, Z., Nam, Y. S., Govindan, R., Rao, S., Chen, J., Katz-Bassett, E., Ribeiro, B., Zhan, J., and Zhang, H. Oboe: Auto-tuning video abr algorithms to network conditions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, Association for Computing Machinery, p. 44–58.

[3] Alammar, J. The illustrated transformer. https://jalammar.github.io/illustrated-transformer/. Accessed: 2022-07-15.

[4] Ba, J. L., Kiros, J. R., and Hinton, G. E. Layer normalization, 2016.

[5] Baevski, A., Hsu, W.-N., Xu, Q., Babu, A., Gu, J., and Auli, M. Data2vec: A General Framework for Self-supervised Learning in Speech, Vision and Language. *arXiv:2202.03555 [cs]* (Apr. 2022).

[6] Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015 ; Conference date: 07-05-2015 Through 09-05-2015* (Jan. 2015).

[7] Bartulovic, M., Jiang, J., Balakrishnan, S., Sekar, V., and Sinopoli, B. Biases in data-driven networking, and what to do about them. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2017), HotNets-XVI, Association for Computing Machinery, p. 192–198.

[8] Beltagy, I., Peters, M. E., and Cohan, A. Longformer: The Long-Document Transformer. *arXiv:2004.05150 [cs]* (Dec. 2020).

[9] Box, G. E. P., and Jenkins, G. *Time Series Analysis, Forecasting and Control*. Holden-Day, Inc., USA, 1990.

[10] Brown, R. *Smoothing, Forecasting and Prediction of Discrete Time Series*. Dover Phoenix Editions. Dover Publications, 2004.

[11] 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).

[12] The caida anonymized internet traces data access. `https://www.caida.org/catalog/datasets/passive_dataset_download/`.

[13] Chen, L., Lingys, J., Chen, K., and Liu, F. Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, Association for Computing Machinery, p. 191–205.

[14] Cornegruta, S., Bakewell, R., Withey, S., and Montana, G. Modelling radiological language with bidirectional long short-term memory networks. In *Proceedings of the Seventh International Workshop on Health Text Mining and Information Analysis* (Auxtin, TX, Nov. 2016), Association for Computational Linguistics, pp. 17–27.

[15] Crawdad. `https://crawdad.org/`.

[16] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv:1810.04805 [cs]* (May 2019).

[17] dhukic, V., Jyothi, S. A., Karlas, B., Owaida, M., Zhang, C., and Singla, A. Is advance knowledge of flow sizes a plausible assumption? In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 565–580.

[18] Dietmüller, A., Ray, S., Jacob, R., and Vanbever, L. A new hope for network model generalization, 2022.

[19] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. *arXiv:2010.11929 [cs]* (June 2021).

[20] Falcon et al., W. Pytorch lightning. *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning 3* (2019).

[21] Fu, S., Gupta, S., Mittal, R., and Ratnasamy, S. On the use of ML for blackbox system performance prediction. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (Apr. 2021), USENIX Association, pp. 763–784.

[22] Gehring, J., Auli, M., Grangier, D., Yarats, D., and Dauphin, Y. N. Convolutional sequence to sequence learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (2017), ICML'17, JMLR.org, p. 1243–1252.

[23] Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, Nov. 2016.

[24] Grus, J. *Data Science from Scratch: First Principles with Python*, 1st ed. O'Reilly Media, Inc., 2015.

[25] Han, K., Wang, Y., Chen, H., Chen, X., Guo, J., Liu, Z., Tang, Y., Xiao, A., Xu, C., Xu, Y., Yang, Z., Zhang, Y., and Tao, D. A survey on vision transformer. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2022), 1–1.

[26] He, K., Chen, X., Xie, S., Li, Y., Dollár, P., and Girshick, R. Masked Autoencoders Are Scalable Vision Learners, Dec. 2021.

[27] Hinton, G., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network, 2015.

[28] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural Comput. 9*, 8 (nov 1997), 1735–1780.

[29] Huber, P. J. Robust Estimation of a Location Parameter. *The Annals of Mathematical Statistics 35*, 1 (1964), 73 – 101.

[30] Jay, N., Rotman, N., Godfrey, B., Schapira, M., and Tamar, A. A Deep Reinforcement Learning Perspective on Internet Congestion Control. In *Proceedings of the 36th International Conference on Machine Learning* (May 2019), PMLR, pp. 3050–3059.

[31] Jog, S., Liu, Z., Franques, A., Fernando, V., Abadal, S., Torrellas, J., and Hassanieh, H. One protocol to rule them all: Wireless Network-on-Chip using deep reinforcement learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (Apr. 2021), USENIX Association, pp. 973–989.

[32] Kairouz, P., McMahan, H. B., Avent, B., Bellet, A., Bennis, M., Bhagoji, A. N., Bonawit, K., Charles, Z., Cormode, G., Cummings, R., D'Oliveira, R. G. L., Eichner, H., El Rouayheb, S., Evans, D., Gardner, J., Garrett, Z., Gascón, A., Ghazi, B., Gibbons, P. B., Gruteser, M., Harchaoui, Z., He, C., He, L., Huo, Z., Hutchinson, B., Hsu, J., Jaggi, M., Javidi, T., Joshi, G., Khodak, M., Konecný, J., Korolova, A., Koushanfar, F., Koyejo, S., Lepoint, T., Liu, Y., Mittal, P., Mohri, M., Nock, R., Özgür, A., Pagh, R., Qi, H., Ramage, D., Raskar, R., Raykova, M., Song, D., Song, W., Stich, S. U., Sun, Z., Theertha Suresh, A., Tramèr, F., Vepakomma, P., Wang, J., Xiong, L., Xu, Z., Yang, Q., Yu, F. X., Yu, H., and Zhao, S. *Advances and Open Problems in Federated Learning*. now, 2021.

[33] Kazemian, P., Varghese, G., and McKeown, N. Header space analysis: Static checking for networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, Apr. 2012), USENIX Association, pp. 113–126.

[34] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings* (2015), Y. Bengio and Y. LeCun, Eds.

[35] Krogh, A., and Hertz, J. A. A simple weight decay can improve generalization. In *Proceedings of the 4th International Conference on Neural Information Processing Systems* (San Francisco, CA, USA, 1991), NIPS'91, Morgan Kaufmann Publishers Inc., p. 950–957.

[36] Kuchaiev, O., and Ginsburg, B. Factorization tricks for lstm networks, 2017.

[37] Measurement lab. `https://www.measurementlab.net/`.

[38] Mao, H., Netravali, R., and Alizadeh, M. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles CA USA, Aug. 2017), ACM, pp. 197–210.

[39] Mitchell, T. M. The need for biases in learning generalizations. Tech. rep., Carnegie Mellon University, 1980.

[40] Montazeri, B., Li, Y., Alizadeh, M., and Ousterhout, J. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, Association for Computing Machinery, p. 221–235.

[41] Neyshabur, B., Bhojanapalli, S., McAllester, D., and Srebro, N. Exploring generalization in deep learning. *Advances in neural information processing systems 30* (2017).

[42] Ng, A. Nuts and bolts of building applications using deep learning. NIPS, 2016.

[43] Nie, X., Zhao, Y., Li, Z., Chen, G., Sui, K., Zhang, J., Ye, Z., and Pei, D. Dynamic tcp initial windows and congestion control schemes through reinforcement learning. *IEEE Journal on Selected Areas in Communications 37*, 6 (2019), 1231–1247.

[44] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035.

[45] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[46] Poupart, P., Chen, Z., Jaini, P., Fung, F., Susanto, H., Geng, Y., Chen, L., Chen, K., and Jin, H. Online flow size prediction for improved network routing. In *2016 IEEE 24th International Conference on Network Protocols (ICNP)* (2016), pp. 1–6.

[47] Riley, G. F., and Henderson, T. R. *The ns-3 Network Simulator*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 15–34.

[48] Robbins, H. E. A stochastic approximation method. *Annals of Mathematical Statistics 22* (2007), 400–407.

[49] Rocketfuel: An isp topology mapping engine. `https://research.cs.washington.edu/networking/rocketfuel/`.

[50] Shaw, P., Uszkoreit, J., and Vaswani, A. Self-attention with relative position representations, 2018.

[51] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res. 15*, 1 (jan 2014), 1929–1958.

[52] Storks, S., Gao, Q., and Chai, J. Y. Recent advances in natural language inference: A survey of benchmarks, resources, and approaches, 2019.

[53] Taylor, W. L. Cloze procedure: A new tool for measuring readability. *Journalism Quarterly 30*, 4 (1953), 415–433.

[54] Tsay, R. S. Time series and forecasting: Brief history and future research. *Journal of the American Statistical Association 95*, 450 (2000), 638–643.

[55] Valadarsky, A., Schapira, M., Shahaf, D., and Tamar, A. Learning to route. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2017), HotNets-XVI, Association for Computing Machinery, p. 185–191.

[56] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems* (2017), I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30, Curran Associates, Inc.

[57] Wettig, A., Gao, T., Zhong, Z., and Chen, D. Should you mask 15% in masked language modeling?, May 2022.

[58] Whittle, P. *Hypothesis Testing in Time Series Analysis*. Statistics / Uppsala universitet. Almqvist & Wiksells boktr., 1951.

[59] Winstein, K., and Balakrishnan, H. Tcp ex machina: Computer-generated congestion control. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, Association for Computing Machinery, p. 123–134.

[60] 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)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 495–511.

[61] Yandex, A. B., and Lempitsky, V. Aggregating local deep features for image retrieval. In *2015 IEEE International Conference on Computer Vision (ICCV)* (2015), pp. 1269–1277.

[62] Yu, Y., Wang, T., and Liew, S. C. Deep-reinforcement learning multiple access for heterogeneous wireless networks. *IEEE Journal on Selected Areas in Communications 37*, 6 (2019), 1277–1290.

[63] Zaheer, M., Kottur, S., Ravanbhakhsh, S., Póczos, B., Salakhutdinov, R., and Smola, A. J. Deep sets. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Red Hook, NY, USA, 2017), NIPS'17, Curran Associates Inc., p. 3394–3404.

[64] Zhang, Q., Ng, K. K. W., Kazer, C., Yan, S., Sedoc, J., and Liu, V. MimicNet: Fast performance estimates for data center networks with machine learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event USA, Aug. 2021), ACM, pp. 287–304.

[65] Zhu, H., Gupta, V., Ahuja, S. S., Tian, Y., Zhang, Y., and Jin, X. Network planning with deep reinforcement learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (New York, NY, USA, 2021), SIGCOMM '21, Association for Computing Machinery, p. 258–271.

Appendix A

NTT training details

We present here further specifics about the choices made during pre-training and fine-tuning our NTT architecture. We also present the common hyper-parameters used for both. In the scope of our project, we do not perform a search to find the best performing hyper-parameters. Our objective is to explore what can be learnt, and not to achieve state-of-the-art results. The hyper-parameters for us are chosen based on Transformers trained in other domains, and with some tweaking, work reasonably well for our use case.

We implement our NTT in Python, using the PyTorch[44] and PyTorch Lightning[20] libraries. We implement our NTT in a Debian 10 environment. For our training process, we use NVIDIA® Titan Xp GPUs, with 12 GB of GPU memory. For pre-training and fine-tuning on the full datasets, we use 2 GPUs with PyTorch's DataParallel implementation. For pre-processing our data, generating our input sliding window sequences and converting our data into training, validation and test batches, we use 4 Intel® 2.40 GHz Dual Tetrakaideca-Core Xeon E5-2680 v4 CPUs and between $60 - 80$ GB of RAM.

Hyper-parameter	Value
Learning rate	1×10^{-4}
Weight decay	1×10^{-5}
# of attention heads	8
# of Transformer layers	6
Batch size	64
Dropout prob.	0.2
Pkt. embedding dim.	120

Table A.1: Hyper-parameters for NTT training

We refer to Table A.1 to discuss our training hyper-parameters. The number of attention heads refers to the number of attention matrices used inside the Transformer encoder layers, which are processed in parallel. In our NTT architecture (Figure 3.4), we have $691K$ trainable parameters from the embedding and aggregation layers, $3.3M$ trainable parameters from the Transformer encoder layers and $163K$ trainable parameters from linear layers in the decoder. We use 4 linear layers in the decoder, with activation and layer-weight normalisation[4] between each linear layer. We also use a layer-weight normalisation layer, as a pre-normalising layer, on the output of the embedding and aggregation. During training, we use a dropout probability[51] of 0.2 and a weight decay[35] over the weights (not biases)[23], in order to prevent overfitting. We use a batch size of

64, to reduce the noise during our training process.

We use the ADAM[34] optimiser with $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 1 \times 10^{-9}$, for our training and the Huber loss[29] function for training loss (A.1), as it is not super-sensitive to outliers but neither ignores their effects entirely. The loss function is computed on the residual i.e. the difference between the observed and predicted values, y and $f(x)$ respectively.

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for } |y - f(x)| \leq \delta, \\ \delta \cdot (|y - f(x)| - \frac{1}{2}\delta) & \text{otherwise} \end{cases} \tag{A.1}$$

We use a warm-up scheduler over our base learning rate (lr) of 1×10^{-4}, as mentioned in the original Transformer paper[56]. We present the governing equation for that as (A.2)

$$lr = d_{model}^{-0.5} \cdot \min\left(step\_num^{-0.5}, warmup\_steps^{-0.5}\right) \tag{A.2}$$

This corresponds to increasing the learning rate linearly for the first *warmup_steps* training steps, and decreasing it thereafter proportionally to the inverse square root of the step number. We used $warmup\_steps = 4000$ and our pre-training data has $\sim 17K$ steps, and our d_{model} is 120.

Appendix B

Learning with multiple decoders

In Section 4.3.3, we evaluated the idea of masking different positions in the input sequence, in order to improve the pre-training phase of the NTT. During this, we realised that with a variable masking approach, it is not always feasible to have a single set of linear layers to effectively act a combined MLP decoder, across all levels of aggregation. We present some further results on using different instances of identical MLP decoders, for different levels of aggregation, which are due to selecting the packet delays to be masked in different ways for pre-training. We summarise our findings in Table B.1.

all values $\times 10^{-3}$ (Masking + MLP instance)	Pre-training (Delay)
NTT: Chosen mask	
From encoded states + 1 MLP decoder	0.063
From encoded states + 3 MLP decoders	0.070
From aggregation levels + 1 MLP decoder	1.31
From aggregation levels + 3 MLP decoders	0.087

Table B.1: MSE on delay prediction across NTT with multiple instances of linear MLP decoders

Based on our experiments, it is evident that we will need different instances of MLP decoders, when we mask across different levels of aggregation, with varying frequency of masking. In the case, where we choose from *the encoded states*, we only choose the packets which are aggregated twice 1/48 of the times, the packets which are aggregated once 15/48 of the times, and hence we mainly choose the non-aggregated packets i.e. 32/48 of the times. In this scenario, it doesn't hurt the performance even if we use a single set of linear layers to extract the learnt behaviour across levels of aggregation, as most of the times, the non-aggregated packets are chosen. In this case, using a single MLP decoder vs using 3 MLP decoders for learning has very similar performance.

When we choose from *aggregation levels*, the situation is very different as every type of aggregation is chosen 1/3 of the times. Effectively this results in the fact, that we mask 1/2 of the packet delays, which are aggregated twice, 1/3 of the times. In this scenario, a single MLP decoder cannot effectively learn across levels of aggregation, all of which are chosen frequently and in this case, using multiple MLP decoders, helps the learning process. A-priori, we do not know what kind of architecture works best for this, and thus we start with the simplest model, where we use three identical sets of linear layers, to match the independent levels of aggregation.

Based on the fact that different aggregation schemes may be used in future version of the NTT,

different numbers of MLP decoders will be needed. One can certainly be sure that, with increasing complexity and aggregation, more complexity will also be required in the decoder architecture, to learn newer kinds of information.

Appendix C

Delay distributions on the multi-path topology

We present further insights in this section, about the individual delay distributions seen on each end-to-end path (from sending sources to each individual receiver) as shown in Figure 4.5. A-priori, we should not assume that increasing complexity in traffic flows on the network, changes the traffic distributions on each individual path. Our NTT learns dynamics only on a single path during pre-training. We hypothesise that this can generalize to topologies with different paths and different dynamics with fine-tuning. For this we should try to check for the case, when these delay dynamics on different paths (affected by queueing delay and link delay) are different.

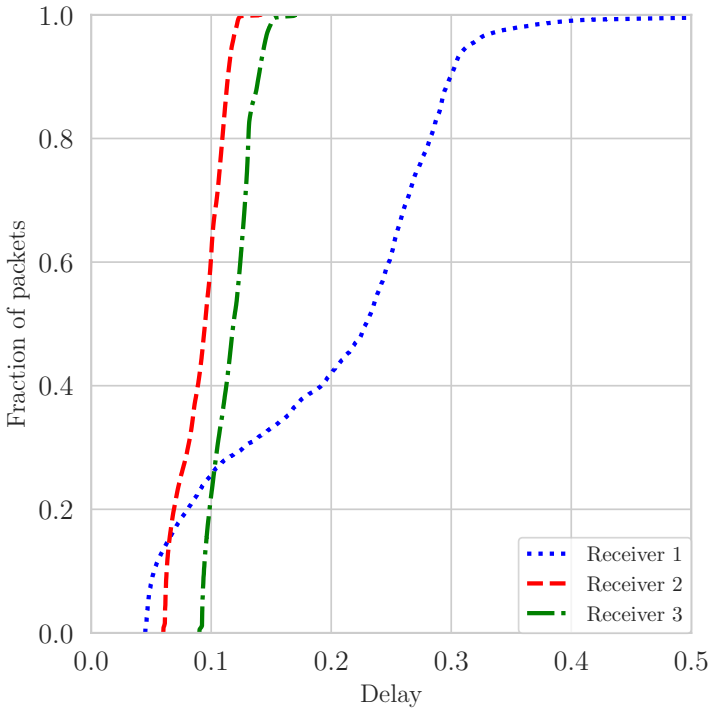


Figure C.1: Comparing delay CDFs across multiple paths

We can see in Figure C.1 that the network dynamics change considerably, as we increase the paths in the network, as we compare the CDFs of the packet delays at all receivers. We observe that

the dynamics change a lot from the path to Receiver 1 to the path to Receiver 2, but in our setup, not so much in the paths to Receiver 2 and Receiver 3. IWe can clearly see from the experimental results in the Section 4.3.4, that the pre-trained NTT generalizes to newer topologies with varying dynamics across different paths. However, for testing the NTT in a more robust manner, it is evident that we need to fine-tune on multiple topologies, with different path dynamics in order to check the true extent of generalization. For the purpose of this thesis, such evaluation is not in the current scope and we leave that to future experiments.

Appendix D

Declaration of Originality

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Advancing Packet-Level Predictions with Transformers

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Ray	Siddhant

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 17th Aug 2022	Siddhant Ray

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.