SCTE CABLE-TEC EXPO FAST FORWARD 2021
VIRTUAL EXPERIENCE
OCTOBER 11-14

2021 Fall Technical Forum
SCTE • NCTA • CABLELABS

Using AI in Network Planning and Operations Forecasting

Petar Djukic
Director AI & Analytics
Ciena Canada
Ottawa ON, Canada
pdjukic@ciena.com

Maryam Amiri
Lead AI Engineer
Ciena Canada
Ottawa ON, Canada
maamiri@ciena.com

Table of Contents

List of Figures

List of Tables

1. Introduction

There are two main sets of costs in network planning that can be reduced through automation and artificial intelligence (AI):

- First, the process today is highly manual. It takes up the time of personnel who could be better utilized if the process was highly automated. Think of network planners with intrinsic knowledge of their network who spend most of their time updating Excel spreadsheets by hand. Their valuable experience could be used much better to maintain the network and to ensure that the customers are happy. ***Automation*** is a tool that can help reduce this set of inefficiencies.

- Second, network planning depends on the ability to forecast network traffic demands. Today this is typically done in an ad hoc fashion where an initial guess of the future network traffic demands is adjusted by essentially a gut feeling from multiple stakeholders. The problem with this approach is that it may result in many inaccuracies in the forecast. Overestimates in the forecast result in too much bought or deployed equipment and then underutilization of network resources. Underestimates in the forecast result in too little deployed equipment, lowered quality of service (QoS) observed by the customer and delayed service deployments, which must wait for equipment to be installed. ***Artificial intelligence*** (AI) is a tool that can reduce this set of inefficiencies.

We note that the two problems feed on each other and prevent either from being solved effectively. The first problem, which is one of automation is more obvious to network operators. As a network equipment vendor, we often hear of the customer woes caused by complicated planning spreadsheets. Through planning network deployments, we also observe first-hand the impact of inaccurate traffic forecasts. The two problems feed on each other as follows: the logic in one direction is “since we have to manually forecast network traffic there is no point automating the rest of the process as there will always be manual steps anyway” and in the other direction “since we manually do planning, there is no point automating the forecasting”. It is also quite possible that network operators are not fully aware of the latest forecasting tools, some of which we talk about in this paper and which can break this cycle of circular logic.

Our hypothesis is that increased automation and the use of artificial intelligence can reduce planning costs, while increasing service provider’s velocity and agility. The focus of this paper is on artificial intelligence and its automation, which should be a medium-term target for the industry. Network planning automation is a near-term topic which many vendors are addressing with their software solutions. There are many vendors advertising their ability to import spreadsheets and incorporate them into automation software, so we will not talk about this topic anymore.

Our main focus is on how AI can be used to automate the process. The goal of using AI technologies is automate as much of the network planning process as possible. Figure 1 shows how this could be done.

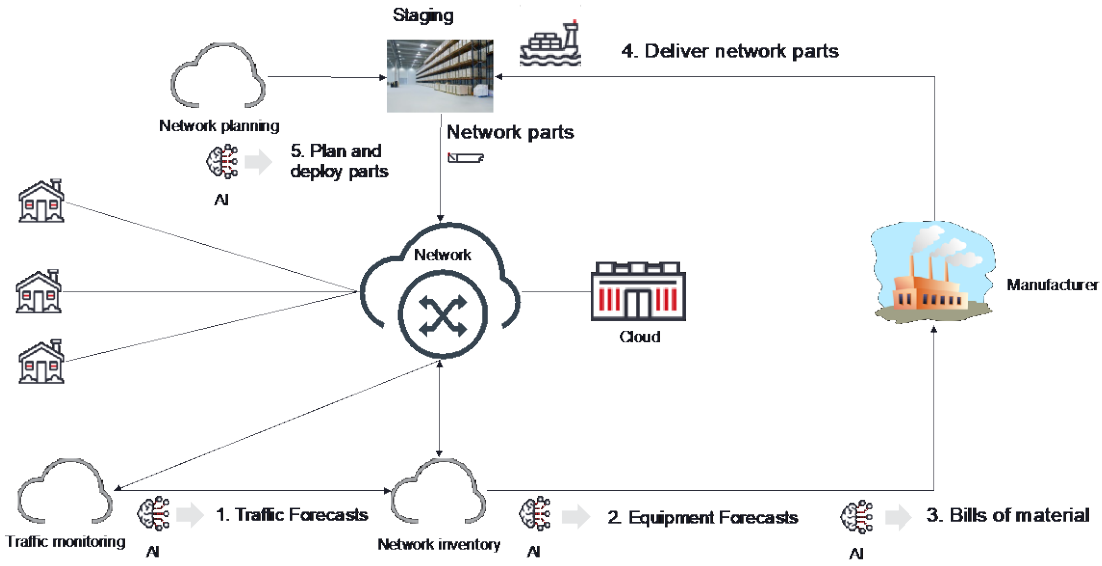


Figure 1 A Vision for a self-planning network

In Figure 1, the network connects the users to the cloud, where most of today's services reside. The traffic monitoring module collects network measurements, such as packet counters and network conditions (e.g., on fibers, or with IPFIX) and uses AI to create traffic forecasts (1). The traffic forecasts are passed to the network inventory module, which uses AI to create equipment forecasts – how much equipment to buy (2). The equipment forecast is passed on to another AI module, which optimizes costs and delivery times for equipment and creates bills of materials, which are sent to manufacturers (3). The manufacturer delivers network equipment to a staging area (4). Meanwhile, the network planning module uses AI to plan network deployments and dispatches technicians to install equipment from the staging area. Ideally, the only manual process is to install equipment and other parts of the process are fully automated with software and AI.

The rest of the paper is organized as follows. We start with a short description of how AI is implemented using deep neural networks (DNNs), following a description of a software architecture which is required to incorporate DNNs into network planning processes. Then we talk about forecasting techniques for network measurements, and we show some performance results using DNNs to forecast network traffic.

Throughout the paper we cite Wikipedia and AI blogs for various AI concepts. While this may appear to not be the most scientifically sound, we found these articles easy to follow and they always link to the more complete computer science papers for the keen reader. There is much DNN jargon used in the paper. We introduce DNN-specific terms in quotes "" to emphasize their jargon origins.

2. Foundations of AI technologies

AI technologies are based on the use of deep neural networks (DNN) for machine learning (ML). Machine learning is a computer science concept in which functional blocks are created by showing the computer examples of correct outputs from inputs, instead of explicitly writing functions that instruct the computer on how to produce outputs from inputs in structured programming (Wikipedia, n.d.). Instead of coding the algorithm, a generic machine learning algorithm is "trained" with examples of what the correct outputs are for given inputs. In recent years, DNN technology has elevated machine learning cognition to the level of human capability. For example, it is now possible to train a DNN-based machine learning algorithm to read a paragraph of text and answer questions about it more accurately than humans, or to categorize x-ray images better than radiologists (Zhang, et al.).

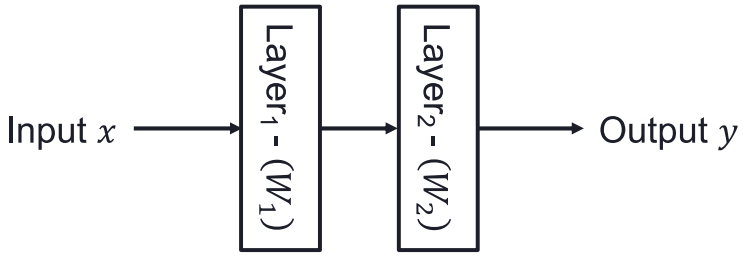
DNN technology is based on basic linear algebra components – matrix multiplication and addition and basic calculus –derivatives. Most of the knowledge required for DNNs has been around for hundreds of years since the time of Carl Friedrich Gauss (Wikipedia, n.d.) and Issac Newton (Wikipedia, n.d.). What is new at this time is that the advances in parallel computing have made it possible to deal effectively with large matrices and train very large DNNs. The most common computing platform are the Graphic Processing Units (GPUs) (Wikipedia, n.d.), which can be used even beyond DNNs, and they are now being complemented with Tensor Processing Units (TPUs) (Wikipedia, n.d.), which are specialized computing units for DNNs. AI accelerators such as TPUs are now found almost anywhere from being embedded in laptops, cellphones, and dedicated data center servers.

DNN-based AI technologies are bringing two main advantages to machine learning. First, as we mentioned DNN-based techniques are starting to perform tasks better than humans. Even though these tasks are limited in nature, it is still impressive to see this, and it opens the question of what other tasks DNN-based ML do better than humans. This is one of the questions we want to answer in this paper. Second, unlike other ML techniques DNN-based ML algorithms are highly automatable. This means that the role of humans in ML is now transferring from traditional manually intensive tasks such as feature engineering, feature selection and tuning of algorithms performed by data scientist, to more traditional software building, which automates these tasks. We explain these concepts shortly. An astute reader will notice that automation of ML is as important as the automation of the network operations to produce a self-planning network. The difference is that in ML this problem is well on its way to being completely solved.

For completeness and interest of the reader we now give a simplified overview of how DNNs make predictions and how they are trained. This is followed by an overview of the different aspects of DNN ML automation.

2.1. How DNNs make predictions

A DNN is a set of algebraic equations that describe how outputs are determined from inputs using matrix operations. A graphical version of the DNN representation is shown Figure 2a, which shows the most basic type of building block for DNNs, known as "dense blocks". The 2-layer DNN shown in the figure is shallow. A typical may have dozens of layers.



a) Graphical Description of a 2-layer neural network

$$y = \max(0, W_2 \times \max(0, W_1 \times x + b_1) + b_2)$$

b) Mathematical Description of the 2-layer neural network

Figure 2 An example 2-layer DNN used for forecasting

The network in shown Figure 2a evaluates an equation involving linear algebra shown in Figure 2b. In this example equation, $W_1 \times x$ is a matrix multiplication (Wikipedia, n.d.) and the max function ensures that the result of all operations is positive. Terms b_1 and b_2 are called bias for the layer. The input to the network is x, while the output is y, so the equation describes the functional block performed by the DNN. The output y is also called a prediction and in the case of forecasting, x is the historical time-series we are trying to predict while y is the future value we need to know for planning purposes. The input x is a mathematical vector whose components are called "features". Each feature is a separate input variable contributing to the output of the DNN. In the case of forecasting, "features" are past samples of the observed network measurements.

The simple set of algebraic transformations in this example is very powerful as it can be shown mathematically that a neural network with enough layers – depth – can approximate any function. For this reason, DNNs are known as "universal approximators" (Hanin & Sellke, 2018).

A pictorial description of a DNN shown in Figure 2a can be translated into the above equation in Figure 2b by an AI engineer and then made into a software program that performs the set of algebraic equations. In practice, the software piece is simple to write using libraries such as TensorFlow (Abadi, et al., 2015) and PyTorch (Paszke, et al., 2019). The DNN can also be exported into the Open Neural Network Exchange (ONNX) (Open Neural Network Exchange, n.d.), which describes the equations and can be loaded into many DNN software frameworks.

Going even further than the simple daisy chain we used, more complex structures could be built by using feedback – Long short-term memory (LSTM) networks and sparse matrices – convolutional neural networks (CNNs). LSTM networks are thought to be good for forecasting as they can model sequential nature of time-series where past values are related to future values. As it turns out, LSTM is not particularly good for network time-series as we show later in the paper. A more promising area for network time-series forecasting is the recent development of matrices that perform inverse fast Fourier transform – IFFT, matrices that perform the inverse

discrete wavelet transform – IDWT and matrices that perform polynomial functions. We talk about these later in the paper.

2.2. How DNNs learn

So far, we described the prediction or inference part of the DNN use. If we know the weights of the DNN (e.g., matrices W_1 and W_2 in Figure 2) then upon receiving the inputs, the set of matrix calculations described by the DNN is performed to determine the outputs. The outputs of the DNN are called the "predictions". This process of making prediction is sometimes called "inference". Weights are determined during a process of training.

For example, if we have the function

$$y = 2x^2 + 3x,$$

we can generate a dataset of training samples shown in Table 1 in the two left-most columns. With the dataset we can use a training function provided in open-source software such as TensorFlow (Abadi, et al., 2015) to determine a set of matrices W_1 and W_2 that result in the best approximation of the function. This function is called "fit" as it fits the weights of the DNN to the dataset during the training. The fit function minimizes the error of the predictions $\hat{y}$ for the dataset compared to actual values in the dataset y. The error can be measured with the Mean Absolute Percentage Error (MAPE) shown in the right-most column of Table 1.

Table 1 Example training dataset for $y = 2x^2 + 3x$

Input x	Actual output y	Predicted output $\hat{y}$	MAPE $\frac{\|\hat{y}-y\|}{y}$
1	5	4.5	10 %
2	14	15.6	11.5 %
3	27	25.4	5.9 %

This simple example may make it seem odd. Why is a DNN learning a known function when we could just be using the function itself? The true power of DNNs comes in when the function is not known but has been observed. In this case, the dataset is a set of observed values coming from the network and the underlying process that models it is not known. For example, the observed values could be packet counters or SNR measurements. In the forecasting case, the training procedure will determine a relationship between the past and future observed values. Once the DNN is trained it can be used for forecasting, by taking in past values and then giving out future values.

The great DNN research achievement in recent years has been to devise a training procedure that uses a set of inputs and outputs of a function to find the set of internal weights W to approximate the function. The training procedure uses "stochastic optimization" whose understanding essentially requires a PhD in mathematics or computer sciences. However, this understanding is not necessary to use DNNs as almost anyone who understands software development can write approximately 10 lines of code to create the DNN and train the function.

The training procedure is iterative and takes in a set of examples of inputs and outputs in batches. For each batch, the training procedure takes in the inputs and generates predictions using the current matrices. The predictions from the DNN are compared with the known outputs to find the error in the predictions (the "loss" function) and this error is used to adjust the weights. The adjustment is usually done using a gradient descent that takes a learning rate as an input and calculates the error of the predictions from the weights and number of predictions. The learning rate determines how quickly the descent happens and how closely to the best fit the training gets. The gradient of the whole DNN is calculated using "backpropagation" (Wikipedia, n.d.), which is an algorithm applied backwards through the DNN to differentiate it. Backpropagation is one example of automatic differentiation using the chain rule (Wikipedia, n.d.).

2.3. Tunning DNN Models

A DNN model is a trained DNN. A single DNN may have multiple models for different versions of the dataset, or different versions of the training algorithms. Each version may have the same structure (number of matrices, size of matrices, and flow through the matrices), but the weights may be different. In a parallel to software development, DNN models have different versions, which presumably improve with higher version numbers. Unlike software, a DNN model is not guaranteed to work well over time, as the inputs may have significant changes in their statistical properties. An example would be traffic demands changing if a new data center peering point is added to the network.

Tuning DNN models has historically been the task of data scientists. The tuning process involves four parts: feature engineering, feature selection, optimization of training hyper-parameters and selection of the DNN architecture. The reliance on data scientists for tuning of DNNs is now waning due to the introduction of automation, as we show in the next section.

Feature Engineering

Feature engineering is the process of modifying input variables to make them better during training and prediction. Historically, this was a very important part of machine learning and data scientists spent a lot of time on it. With the improvements in DNN technology, which during training learns the best representation of input variables, this process has become almost irrelevant. However, it is still important to scale the input and output variables to small range (typically between 0 and 1) to avoid numerical issues.

Feature Selection

Feature selection removes unimportant features. As the number of features increase so does the size of the DNN as each of the weight matrices needs to have the width of the vector it is multiplied with. A larger DNN size results in a longer time to make a prediction (and therefore to train the network). Most importantly, larger size means that the DNN training requires a larger dataset due to the "curse of dimensionality" (Wikipedia, n.d.). For example, if a DNN requires 300 samples per feature to be trained, adding 10 new features means that we need to have available another 3000 new training examples to get the equivalent performance. As the training data used for forecasting accrues historically, adding 10 new features to the inputs means that more measurements are required to train the DNN. Getting an extra 3000 new 15-minute

network samples for training may take as long as 4 weeks, so reducing training dataset size is a very important problem.

Feature selection is based on the premise that not all DNN inputs contribute equally to the output of a DNN. The basis of this premise is that during training, DNNs use the statistical correlation between input and output variables to deduce the best weights. Input variables uncorrelated to the outputs are not needed at the input to make predictions. Over the years, many feature selection approaches have been developed and this process can be automated through a search of required features (SciKit Learn, n.d.).

Selection of training hyper-parameters

Recall that during training input examples are grouped into *batches*, and that for each batch a gradient of the DNN is calculated using backpropagation and applied to the weights with a *learning rate*. The weights of the DNN are called "parameters" as they are determined during training, while the batch size and the learning rate are "hyper-parameters" as they are inputs to the training procedure.

Selecting the right batch size and learning rate are the easiest way to improve the performance of DNN during training. The process used for this is called hyper-parameter optimization or hyper-parameter tuning (Brownlee, n.d.) and is highly automatable and easily parallelized.

In the rest of the paper, we do not distinguish between training and hyper-parameter tuning. In practice, they are often combined into a single procedure.

Selection of a DNN architecture

The simple DNN example in Figure 2 uses two matrices of matrix weights W_1 and W_2, which are determined during the training process and tuned during the hyper-parameter search. However, the choice of the number and size of matrices may not be obvious for each data set.

At least two architectural parameters are unknown for even the simple example in Figure 2: the number of matrices and the size of each matrix. We used 2 matrices, which was easier to explain in this paper, but a typical DNN may have many more layers than that. The height of each matrix is also not readily known as only the width of each matrix is known (the height of the previous matrix). For more complex internal structures it gets complicated in terms of how the architecture is selected, and there is even an area of DNN techniques dealing with measuring the difference between architectures, called ablation (Wikipedia, n.d.).

To find the best architecture, in parallel to the hyper-parameter a network architecture search (NAS) (Wikipedia, n.d.) is also needed. The NAS space is much larger than the space of hyper-parameters, so this is much bigger problem to automate effectively. Typically, the search is done in unique directions, for example one direction could be a daisy chain of dense layers (shown in Figure 2a), while another direction may be a network with daisy chains of LTSM layers. In each direction the NAS is restricted to the number of layers and the height of each layer. Recently, this has been generalized in open-source software (OSS) by Google's Model Search (Google, n.d.).

3. Operationalizing DNNs with AI Software

We now pivot to perhaps a more interesting set of topics for network operators – how DNNs are operationalized with software. The main set of software available is free OSS. The AI software stack has evolved over the last 3 years in conjunction with the developments at the Cloud Native Computing Foundation (CNCF) (Cloud Native Computing Foundation: Building sustainable ecosystems for cloud native software, n.d.). The two main drivers are TensorFlow (Abadi, et al., 2015), which implements DNNs, and KubeFlow (Kubeflow: The Machine Learning Toolkit for Kubernetes, n.d.), which is a set of Kubernetes services used to create AI-specific distributed applications. Both OSS projects were initiated and are still strongly supported by large cloud providers (e.g., Google).

3.1. Microservices

The AI software stack is based on the concept of "microservices" (Wikipedia, n.d.), which are meshed with networking into distributed AI applications. This follows today's architectural patterns, where distributed applications based on microservices underpin today Software-as-a-Service (SaaS) software delivery model (Wikipedia, n.d.).

Microservices are implemented with Linux containers, which group Linux processes to have common permissions and resource limits. Communications with microservices are implemented with networking and commonly with a higher layer protocol such as the Hyper-text Transfer Protocol (HTTP) implementing a Representational State Transfer (REST) architectural style (Wikipedia, n.d.). The microservices are also called RESTful as they are assumed to not hold state between subsequent REST application programming interface (API) calls. Each micro-service exposes its API through Uniform Resource Locators (URLs) corresponding to each of its available functions. Microservices in the same distributed application communicate through networking, so they are not guaranteed to run on the same server, or even in the same datacenter.

Microservices can be combined into distributed applications, where there may be layers implementing different functionalities (e.g., the web interface layer and the database layer). To improve security inside the distributed applications, microservices-based distributed application use a "service mesh" (Wikipedia, n.d.) – an overlay topology interconnecting the microservices in a security conscientious way. As the RESTful communication approach does not provide stateful transfer of data, a messaging bus (service) may be employed to simplify communications between microservices in the same distributed application.

The distributed microservices application architecture is very common in cloud applications, and it has been driven by OSS delivered through the CNCF (Cloud Native Computing Foundation: Building sustainable ecosystems for cloud native software, n.d.). The key contribution to CNCF came from Google in the form of Kubernetes (Kubernetes: Production-Grade Container Orchestration, n.d.), which is container resource orchestrator. Most of the work of the CNCF revolves around building software required to make distributed applications managed by Kubernetes easy to make.

3.2. DNN models as microservices

A trained model is described with an ONNX file, which contains the description of the DNN layers and the weights of each layer. ONNX files can be read by all major DNN software libraries and the used to make predictions. An ONNX file does not make predictions, it only describes a network and its weights. When loaded into ONNX hosting software, the software provides a way to make predictions with the stored DNN model. Inside software, a DNN model is used with a function "predict", which takes as an input features x and outputs an estimate $\hat{y}$.

The functional DNN model fits with the RESTful approach of stateless services. The DNN model does not hold state between subsequent prediction calls. The model can be served on a unique URL with an HTTP post request carrying x and the serving microservice returning the prediction $\hat{y}$. This serving functionality is available in all major DNN software distributions. For example, TensorFlow Serving (Tensorflow: Serving Models, n.d.) and TorchServe (PyTorch: TORCHSERVE, n.d.), provide generic serving containers, which can load a DNN model from an ONNX file and then provide a service access point for the model's predict function. In a distributed application, a serving container becomes is a microservice serving multiple models, each with a unique URLs.

3.3. AI architecture

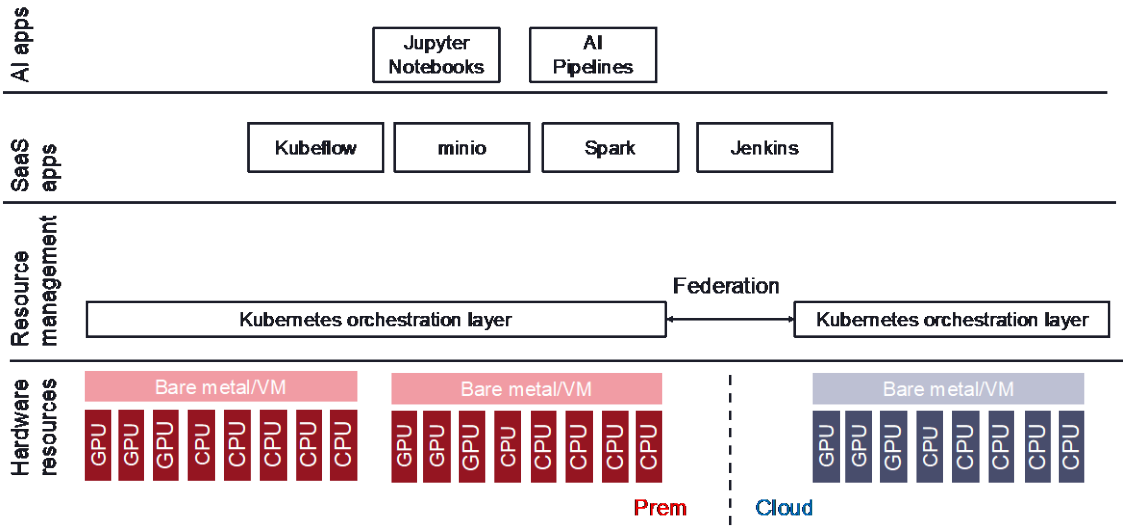


Figure 3 Example AI software stack

The AI software stack is shown in Figure 3 and contains several microservices-based distributed applications running over a distributed hardware architecture. All architectural components in the figure are OSS and are taken from CNCF and TensorFlow family of software. The main AI OSS component, KubeFlow, is spearheaded by major cloud companies, who use it in their cloud and are the basis of their automatic ML (AutoML) offerings. Only the knowledge of Kubernetes is required to setup and maintain the software. It does not take very long to install and set up the stack, or to apply software upgrades. In our opinion, building AI software without using the

Kubernetes and KubeFlow OSS ecosystem would be a strategic mistake, which may result in much unneeded development efforts and may likely result in a complete overhaul of AI software architecture a few years later.

3.4. Architectural Layers

The "hardware resources" are shown at the bottom of Figure 3. They may reside in a private cloud, on the premise close to networking equipment (the edge cloud), or in the public cloud. Hardware resources are made available through the Kubernetes API. We note that the hardware resources are not homogeneous or equally distributed. For example, there would be many more compute and storage resources in the cloud then on the premise ("infinitely" so). The Kubernetes resource manager uses the resources according to their cost and latency requirements of the distributed application. Specifically, one would expect to use edge or network resources first if they are available, to achieve best latency, and spill over other less latency constrained workloads into the cloud.

The "resource management" layer in Figure 3 is a federation of Kubernetes container orchestrators, which manages the hardware resources across multiple clouds. There is a Kubernetes instance associated with each separate set of hardware resources (e.g., cloud or edge). The Kubernetes orchestrator keeps track of available hardware resources and allocates the resources requested for each microservice and spins up the microservice on those resources. Separate instances of Kubernetes can be federated to allow for a seamless use of resources regardless of their location. Kubernetes also provides services useful for a microservices-based architecture: a domain name service (DNS) which maps service URLs to IP addresses, load balancing, automatic scaling based on measurements of service latency, monitoring of microservices health, and restarting them when necessary, and a global registry implemented as distributed key-value store. In terms of resource separation and security, Kubernetes provides namespaces which are logically separated groups of microservices.

The "SaaS apps" layer in Figure 3 is a set of distributed applications installed on top of the Kubernetes resource manager, which are required to build distributed AI. For illustration purposes, we show some of the more useful services for AI: MinIO (MinIO: Object Storage for the Era of the Hybrid Cloud, n.d.) can used to store datasets, by providing an S3 (Amazon S3: Object storage built to store and retrieve any amount of data from anywhere, n.d.) compatible object storage, with the ability to store objects (files) transparently on the local servers or in the private or public cloud; Spark (Apache Spark: Lightning-fast unified analytics engine, n.d.) is a distributed in-memory analytics engine capable of processing vast amounts of data, so it can use to process datasets; Jenkins (Jenkins: build great things at any scale, n.d.) is an integration and delivery automation software; and KubeFlow is an AI pipeline orchestrator and DNN model tracker. A Kubernetes cluster may have many other services co-existing with these, depending on how it is used.

The "AI apps" layer is at the top level of the AI software stack in Figure 3. AI apps are built using the KubeFlow SaaS layer. KubeFlow provides facilities for launching of AI specific containers hosting Jupyter Labs Notebooks (Jupyter: Project Jupyter exists to develop open-source software, open-standards, and services for interactive computing across dozens of

programming languages, n.d.) and for creation of AI pipelines, which are distributed applications used for training and validation DNN models. KubeFlow also contains services for tracking and delivery of DNN models.

3.5. AI Pipelines are DNN model factories

AI pipelines are dynamic distributed applications that train DNN models. Each model has its own pipeline, which includes data processing, training with NAS and hyper-parameter selection, and validation steps. Figure 4 shows an example AI pipeline. A pipeline is specified using KubeFlow's domain-specific language (DSL), which describes the containers to be used at each stage of the pipeline. In the example, there is a container for processing data, container to train a model and a container to test a model. When the pipeline is started, Kubeflow spins up the containers in the order specified and ensures that each part of the pipeline finishes before containers for the next part of the pipeline are spun up.

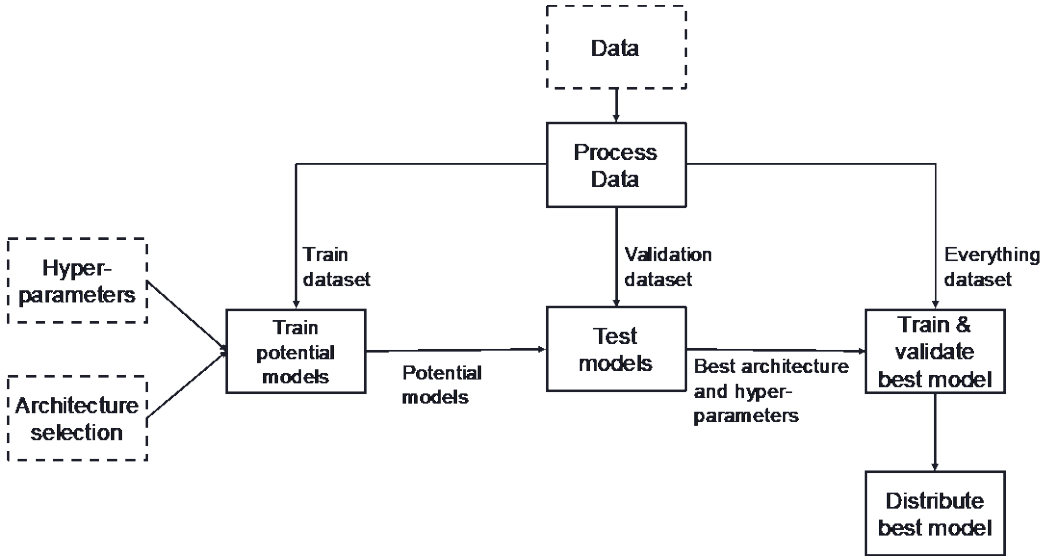


Figure 4 An Example AI pipeline

In the example, data is input to the "process data" step, which requires KubeFlow to provide the data to the process data container, which it also spun up for the pipeline. In practice, data may be in an S3 bucket stored locally or in the cloud. This data processing step may use SaaS services of Spark to process a large volume of data. Note that the data is split into a training and testing dataset to follow common machine learning methodology. The test dataset is provided to the training step, which uses NAS and hyper-parameter search to train many DNN models. This part of the pipeline uses Katib (Kubeflow, n.d.) to create many instances of the same training container with different inputs and runs them in parallel on the Kubernetes cluster. Potential models are evaluated using the validation step. The models are ranked based on the performance of the loss function (error) as shown in Table 1. The DNN model with the lowest error corresponds to the best architecture and hyper-parameters for the training dataset. These are passed to an instance of the training container, which trains the best model with all available

data. The model is validated using hold out data from the dataset. The final step is to distribute and serve the best model.

Figure 5 shows the relationship between AI pipelines and the regular network applications. On the left side of the figure, AI pipelines are a special type of distributed application, used specifically for DNN model search with NAS and hyper-parameter tunning. In our software stack, AI pipelines exist in their own Kubernetes namespace. Each pipeline implements a specific AI use case. For example, network traffic forecasting, which is the topic of this paper, is one use case, which is quite different from other use cases. We show two other use cases in Figure 5: an AI use case that trains a DNN for optical signal-to-noise SNR calculations; and an AI use case that trains a DNN model for network design and planning. The latter may be a DNN model implementing a reinforcement learning approach for path selection in the network. The output of an AI use case is a DNN model. As the DNN models are trained in a namespace different from where they run, then stored in a DNN model registry, which is accessible from all namespaces.

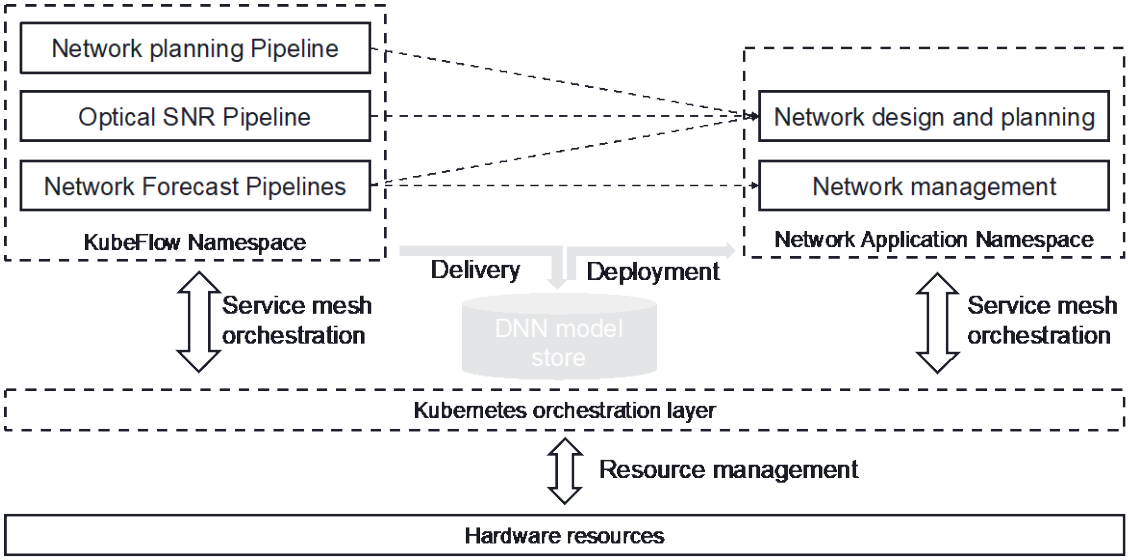


Figure 5 AI pipelines and other distribute network applications

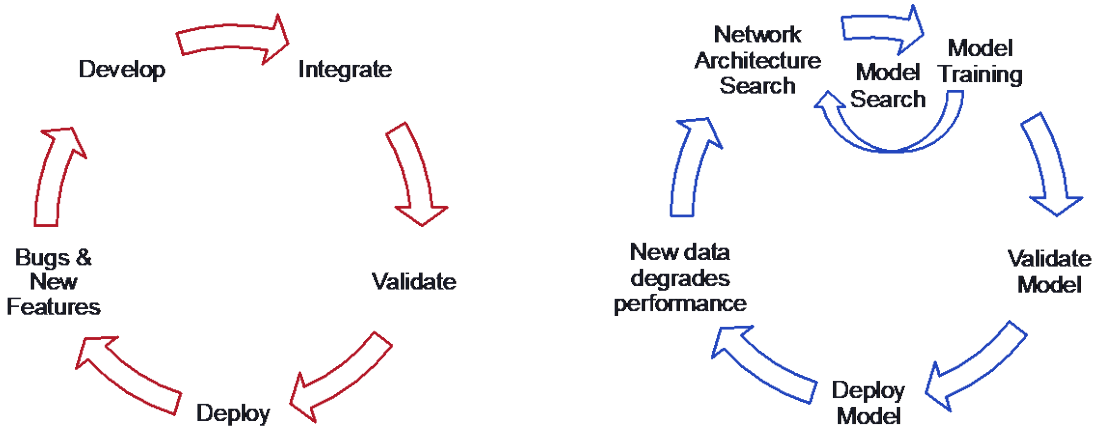
The right side of Figure 5 shows the network applications using the DNN models trained by the pipelines on the left side. The dashed arrows indicate where pipelines deliver models. Note that the network forecasting is many applications, as forecasting could be done at different scales for planning, or management. The model is delivered to the DNN model store and then it is deployed in the corresponding network application.

In the example, we assume that the network applications are hosted on the same Kubernetes cluster. This could be done in separate namespaces as shown in the figure to assure that the network applications are secure and have guaranteed resources. The figure shows several pipelines, the “network planning pipeline”, the “optical SNR pipeline” and the “network forecast pipeline”. Each pipeline produces a DNN model for a specific purpose and with a separate

dataset. To produce the model, pipelines may use the same pipeline template (e.g., template in Figure 4), or they may have different pipeline templates. For example, one of the pipelines may use transfer learning (Wikipedia, n.d.), while other pipelines may use reinforcement learning (Wikipedia, n.d.) to produce their DNN models.

3.6. The MLOps Cycle

The architectural relationships in Figure 5 create an opportunity for MLOps. MLOps (Google) are the equivalent of DevOps (Wikipedia, n.d.) for AI. DevOps is generally accepted way of creating cloud-based software application with a tighter integration of software development and delivery, than what has been done historically. Integral to DevOps is the concept of combined Continuous Integration (CI) and Continuous Delivery (CD). The process is typically done with an automation tool like Jenkins. The CI/CD process is shown in Figure 6a. As the code is being developed, it is automatically tested and integrated and then validated. Testing and integration are done in the development environment, while the validation is done in the testing environment. If the new software is compliant with validation tests it is transferred to the production environment. Even in the production environment the software can be tried out before full deployment. This can be done with Kubernetes, which is instructed to only deploy some portion of the newly created containers into production and load-share application between the new and old containers. Only if the new containers perform satisfactorily, the new containers replace all the containers.



a) DevOps Cycle with CI/CD b) The MLOps Cycle

Figure 6 DevOps vs. MLOps

DevOps are enabled by the ability to automate testing, integration and validation and then automatically deliver containers to the cloud. This ability is right now coming directly from the use of microservices and Kubernetes, combined with CI/CD automation. Microservices architecture allows incremental upgrades (one micro-service at a time). Kubernetes provides a platform to test microservices through namespaces. For example, with Kubernetes it is possible to easily create namespace where different versions of the same microservices are logically separated on the shared hardware and incremental testing of new features and bug fixes can be

done. Kubernetes also provides facilities to gradually upgrade a microservice in production by gradually replacing copies of old versions with new upgraded ones and to monitor the new versions and to roll them back if problems are noticed.

The MLOps cycle is shown in Figure 6a. This is the equivalent to CI/CD for DNN models, with some fundamental differences. The first difference is that the DevOps cycle is triggered by bugs in the code or new features, while the MLOps cycle is triggered by new data, which degrades the performance of the existing DNN model. The second difference is that the DevOps cycle starts by development of the code, while the MLOps cycle starts with a network architecture search and model training, which includes hyper-parameter optimization. Instead of coding a new DNN model, the new model is found automatically using new data. Validation and deployment parts of the cycle are essentially the same. The new model is packaged as a microservice and is updated in the service mesh just like any other microservice.

3.7. Automatic Machine Learning (AutoML)

The combination of AI pipelines with NAS and hyper-parameter search is known as automatic machine learning (AutoML). The search for the best combination of data processing, network architecture and hyper-parameters is completely automated and the DNN model factory in Figure 5 can produce DNN models without human input.

While this may sound magical as the data scientist expertise and experience is greatly reduced and removed from the ML process, we note that there are several points to AutoML that require some human input:

- First, the NAS and hype-parameter space may be quite large. This means that the search space could be so large that the time to create a new model may exceed operational needs. A data scientist or an AI Engineer would be involved to decide on the *reasonable* search space that may generate *good* DNN models.
- Second, model performance in production must be monitored and evaluated as data coming from the network changes over time. A sufficient change in the network data will trigger the MLOps cycle. However, a data scientist still needs to monitor the performance and its trends and make judgment calls when to trigger the MLOps cycle or debug DNN model performance for causes of degradation.
- Third, the role of AI models in the distributed network applications needs to be well understood. This requires a human contribution in understanding the domain of application and to design the system that automates downstream actions of network operators.

So, while data scientists will still be needed in the world of AutoML, their role in the process will change. Instead of each data scientists training a single model over a period of weeks or months, that data scientist will be able to train hundreds of models in a day and debug the select few for performance issues. Data scientists may also take on the role of a translator (Analytics translator: The new must-have role, n.d.) to bridge the gap between business needs and AI technological capabilities.

4. Forecasting the Network Demands with Artificial Intelligence

So far, we have talked about existing and soon-to-be-available AI architectures and components. This section talks specifically about how DNNs can be used for forecasting in the network.

Forecasting is the process of taking in historical values for a network measurement and producing estimates of future values. Network measurements are stored in databases as time-series, "series" is used because measurements are collected uniformly, so only a sequence of measurements is required and time can be calculated by a position in the series.

Several kinds of forecast are possible, depending on what the inputs and outputs are:

- Single-variate forecasts are for time-series containing a single measurement, for example packet counts from a single router link.
- Multi-variate forecasts are for time-series containing multiple measurements, for example packet counts from multiple links on the same router.
- Single-step forecasts produce the next value in time, so if packet counters are collected every 15 minutes, a forecast at 8:00AM will be for the packet counter at 8:15AM.
- Multi-step forecasts produce a sequence of future values, so if packet counters are collected every 15 minutes, a forecast at 8:00AM can be for the next day and produce 96 values (for every 15 minutes in a 24-hour period).

There are only 4 valid forecast types, given the two kinds of inputs and outputs.

The content of this section is best suited for single-variate forecast of single-step or multi-step kind. However, we also show results for multi-step forecasts.

4.1. Forecasting network traffic

Forecasting approaches are based on behavioural models of underlying processes that reflect in the data. For example, some generalities about human generated data are almost always true: human activity almost always increases, and human activity is seasonal. The growth in human activity is especially reflected in the growth of Internet traffic, which seems to only go up (Cisco, n.d.). There is also seasonality in the Internet traffic (Hen & Karlesson, 2019) reflected both in which direction the network traffic flows and in the volume of traffic. For example, the difference is pronounced during a single day with high business traffic during work hours and low business traffic during the evening and high entertainment traffic during the evenings, which are also temporarily different across the world. Generally, seasonality is also noticeable during different days of the week with weekends and holidays showing high consumption of entertainment and low business traffic.

4.2. Traditional forecasting approaches

A generally accepted forecasting approach is to decompose the time-series and extract the trend and seasonality components (Hyndman & Athanasopoulos). The reminder of the time-series includes noise, which needs to be smoothed out. Figure 7 shows the decomposition process. The input is historical time-series. The trend is extracted first, followed by the seasonality. The remainder of the time-series is smoothed. The trend can be used to make long-term forecast (in the order of months, quarters, and years), while the seasonality can be used to make forecasts for

repeating patterns in the time-series (daily, weekly). The smoothed remainder can be used to make short-term forecasts.

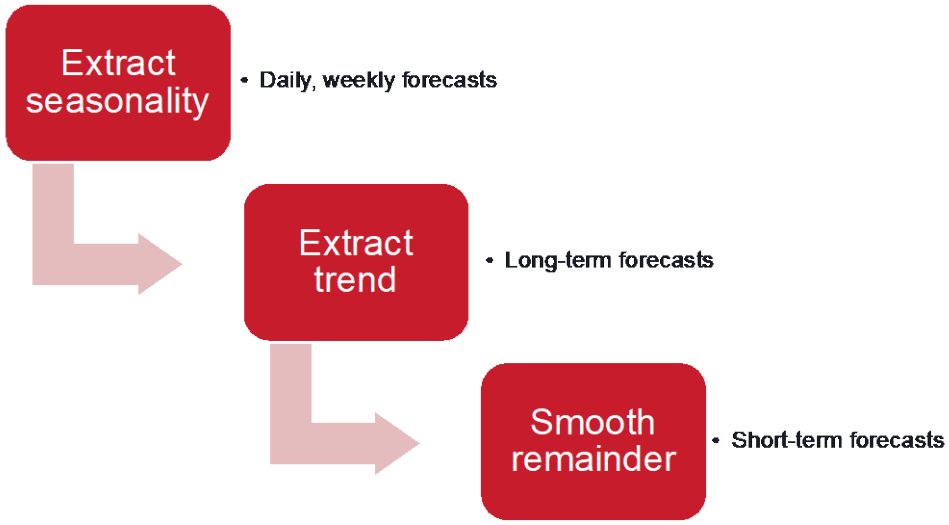


Figure 7 Forecasting time-series decomposition

Traditionally, time-series decomposition has been a highly manual process. The time-series structure is not known before analysis done by someone with a lot of forecasting expertise. The decomposition can be done automatically using one of the newer tools (Prophet: Forecasting at Scale), however a forecasting expert should still ensure that the automation has gone smoothly as the forecasting model in (Prophet: Forecasting at Scale) makes very specific assumptions about the time-series. The forecaster makes an informed decision for each of the time-series components trend, seasonal and smoothing. A typical forecasting expert works with a small dataset and uses judgment in deciding which forecast makes the most sense. For example, forecasting quarterly GDP over the last 50 years only has 200 points and a forecasting expert may use their knowledge of economics and information available outside of the time-series to decide if GDP will go up or down in the next quarter. Compared to network data, this is very small dataset – there are close to 200 data points in two days of 15-minute bins. So, some of the analysis done by traditional forecasting technique doesn't translate well into the networking domain.

In terms of time-series feature, an expert forecaster may use some of the following approaches:

- Seasonality can be estimated in many ways including taking averages at repeating time, e.g., by find an average traffic on Monday 9:00AM-9:15AM, by isolating data in this repeating period of time and then averaging, or with more complex approaches such as estimating the periodicity in the frequency domain.
- Trend is typically chosen as the best line that fits through the data. To make things simple to understand, a forecaster usually picks linear or exponential trend (Wikipedia, n.d.), but more complex methods such as piece-wise linear trends (Hyndman R. J.) are also possible.

- Smoothing is used to remove the noise from the time-series. A familiar from of smoothing may be the moving average (MA) smoother available on stock tracking platforms (Ermey, 2019). The moving average is also statistically the most likely estimate of the next value in the time-series. More sophisticated smoothers also exist, namely the auto-regressive moving average (ARIMA) family of estimators (Wikipedia, n.d.), which model temporal relationships between samples of a time-series. Some forecasters feel confident in using
  ARIMA forecasters for multi-step forecasts, however assumptions on the underlying data should be checked before getting overconfident with this method.

It is important to take stock of the state-of-art in existing forecasting approaches: they are highly dependent on matching the model in the forecasting algorithm to the time-series, because historically there wasn't that much data available for forecasting. Both reasons are why human intervention is required for traditional forecasting approaches.

As we have shown earlier, one of the advantages of using DNNs is that they can learn the best model for a time-series, given enough data. In the networking use cases, there is enough data. We now go over some of the approaches that can be used to automate forecasting with DNNs.

4.3. Forecasting with DNNs

Probably the most accepted DNN forecasting approach is to use recurrent neural networks (RNNs) (Wikipedia, n.d.). RNNs use recursion to pass data from the previous step to the current step as shown in Figure 8. In the figure the DNN block is identical throughout the network, meaning that recursion happens, and the last output is the function of all inputs. The RNN structure models time dependencies in the time-series. The block could be made from anything, but a popular structure long-term short memory (LSTM), which is numerically stable.

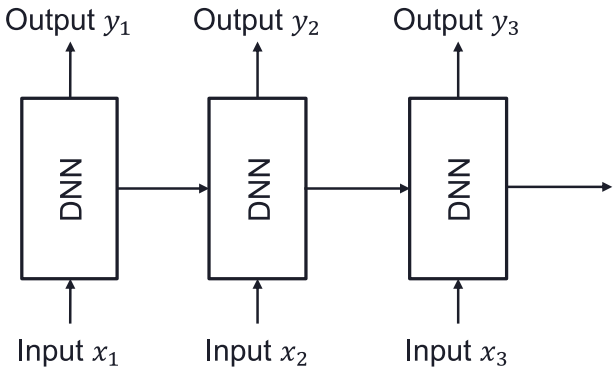


Figure 8 Recursive Neural Network

The assumption of the recursive relationships of time sample is reminiscent of that ARIMA forecasting models. As we already mentioned, this approach is valid for short term forecasts (smoothing) after trend and seasonality have been removed. The assumption of the recurring

relationships in the time-series isn't always true, or important for time-series forecasting. The more important assumptions are around trend and seasonality, which should be accounted for.

Recently, a residual network structure (Wikipedia, n.d.) was used to model this behaviour. In a residual network, it is possible to have skip connections, that allow addition or subtraction of various output blocks. The N-BEATS doubly residual network architecture (G, n.d.) for time-series forecasting is shown in Figure 9. Following from left to right, on the bottom, it can be seen that what is happening in the network is a trend estimate being determined first and then being subtracted from the input. Then the seasonality estimate is determined and subtracted from the "residual" of the input (input with trend subtracted). Finally, the residual of both of those is smoothed out. On the top of the network, the outputs of the three estimates are combined for the forecast.

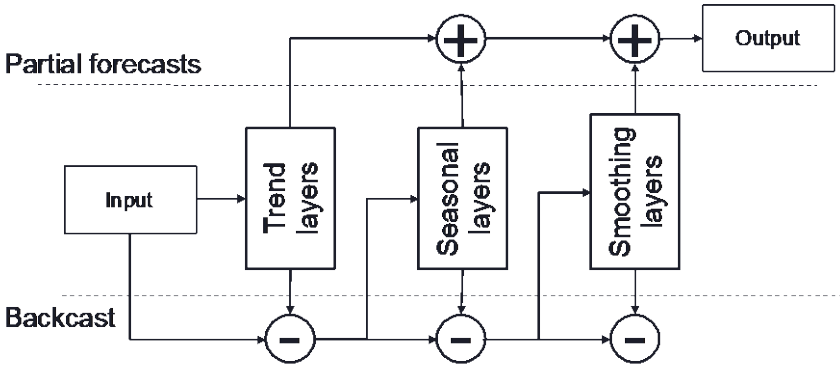


Figure 9 Time-series decomposition

As an example, we used the NBEATS (G, n.d.) network and compared it to the winner of the M4 forecasting competition (The M4 Competition Team, n.d.) dataset. The winner of the competition was a modified LSTM, which was used to win the competition and it required human intervention to perform well. The M4 forecasting competition dataset has over 100,000 financial and manufacturing time-series. Algorithms are compared against a simple baseline, which is used to ensure that there is something to learn in the dataset. The baseline algorithm uses the last seen value as the forecast of future values. It may be hard to believe, but this simple baseline is very hard to beat on real-life datasets.

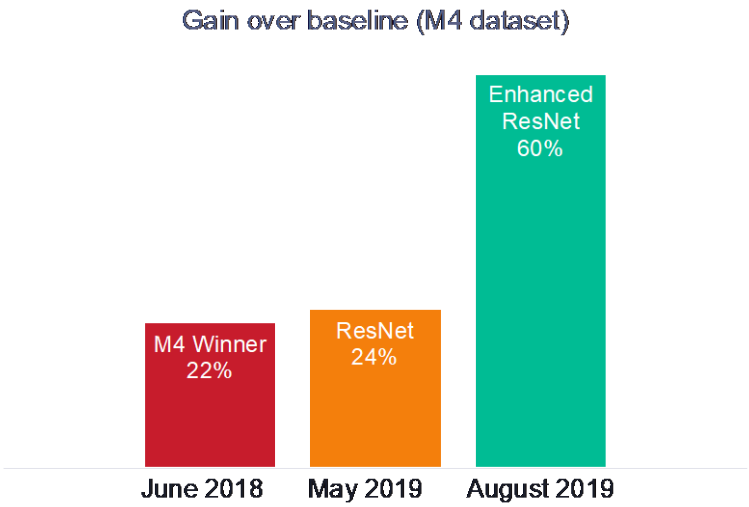


Figure 10 Performance comparison of DNN approaches (public dataset)

In the Figure 10, NBEATS is labeled as ResNet. The relative gain is measured on the error of one method over the other method. So, 22 % gain of the M4 Winner over the baseline means that the error (MAPE) of the winner is 22 % less than the winner. It should be observed that NBEATS improves upon the M4 Winner, but not by a lot. Based on our observations of the NBEATS performance, we developed a new ResNet based algorithm and it shows an improvement of 60% over the baseline (labeled Enhanced ResNet). As far as we can tell, the Enhanced ResNet is the world's best forecaster as it works better than the M4 competition winner and the next best forecaster.

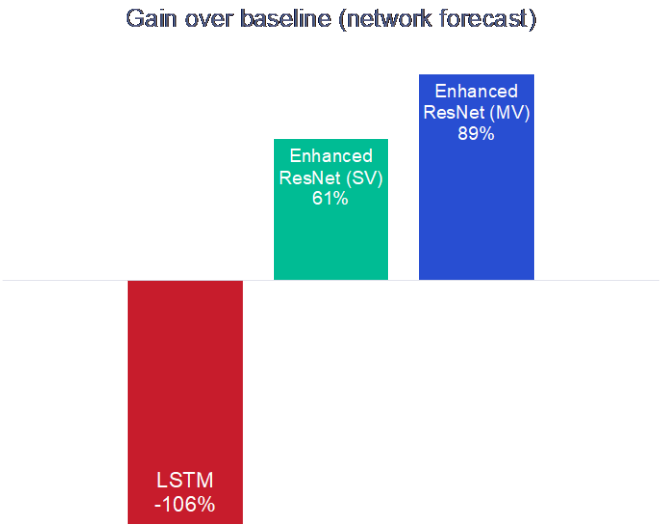


Figure 11 Performance comparison of DNN approaches (network dataset)

Real-world network datasets are quite different from the financial series in the M4 dataset. For the most part, financial time-series are smooth and without sudden changes. On the other hand, network time-series often have unexpected steps and spikes. We tested the LSTM and Enhanced ResNet algorithms on a network time-series in our possession. The data was collected from a large service provider. The results are shown in Figure 11. Note that LSTM does not work well on this dataset. It performs much worse than the baseline. We are confident this is because LSTM assumes too many dependencies on time, so it doesn't work when the time-series have abrupt changes.

On the other hand, the Enhanced ResNet algorithm (shown as "Enhanced ResNet (SV)") works much better than the baseline. Furthermore, the Enhanced ResNet algorithm can be extended to forecast from multi-variate inputs (shown as "Enhanced ResNet (MV)"). Network time-series have dependencies on other time-series, so it is advantageous to combine them during the forecasting process.

5. Summary

This paper has talked about forecasting in the context of network operations. We have made the argument that network operations, especially its planning functions, need to automate all parts of their processes and that this cannot be done without automated forecasting. We have shown how automated forecasting can be done with DNNs and AutoML. We have also used network time-series from an actual network to show the power of forecasting with DNNs.

We believe that DNNs and AutoML have the potential to revolutionize the network planning process and make it more accurate and less costly than today.

Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
CI/CD	Continuous Integration (CI) and Continuous Delivery (CD)
CLI	Command-line interface
CNN	Convolutional Neural Network
CNCF	Cloud Native Computing Foundation
CPU	Central Processing Unit
DNN	Deep neural network
DNS	Domain Name Service
DSL	Domain-specific language
GPU	Graphic Processing Units
IDWT	Inverse Discrete Wavelet Transform
IFFT	Inverse Fast Fourier Transform
IGP	Interior gateway protocols
IP	Internet Protocol
HTTP	Hyper-text Transfer Protocol
IPFIX	IP Flow Information Export
LSTM	Long short-term memory
MAPE	Mean Absolute Percentage Error
ML	Machine Learning
NAS	Network Architecture Search
ONNX	Open Neural Network Exchange
OSS	Open-source software
PoP	Point of Presence
RCA	Root Cause Analysis
RNN	Recurrent Neural Network
SaaS	Software-as-a-Service
S3	Simple Storage Service
SP	Service Providers
TPU	Tensor Processing Unit
URL	Uniform Resource Locators

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., . . . Jozefowicz, R. (2015). *TensorFlow: Large-scale machine learning on heterogeneous systems.* Retrieved from Software available from tensorflow.org: https://www.tensorflow.org/

Amazon S3: Object storage built to store and retrieve any amount of data from anywhere. (n.d.). Retrieved June 5, 2021, from https://aws.amazon.com/s3/

Analytics translator: The new must-have role. (n.d.). (Harvard Business Review) Retrieved June 9, 2021, from https://www.mckinsey.com/business-functions/mckinsey-analytics/our-insights/analytics-translator

Apache Spark: Lightning-fast unified analytics engine. (n.d.). Retrieved June 5, 2021, from https://spark.apache.org/

AWS. (n.d.). *Amazon S3: Object storage built to retrieve any amount of data from anywhere.* Retrieved 07 21, 2021, from https://aws.amazon.com/s3/

AWS. (n.d.). *AWS Simple Monthly Calculator.* Retrieved from https://calculator.s3.amazonaws.com/index.html

Brownlee, J. (n.d.). *Hyperparameter Optimization With Random Search and Grid Search*. Retrieved June 2, 2021, from https://machinelearningmastery.com/hyperparameter-optimization-with-random-search-and-grid-search/

Cisco. (n.d.). *Cisco Annual Internet Report (2018–2023) White Paper*. Retrieved June 9, 2021, from https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

Cloud Native Computing Foundation: Building sustainable ecosystems for cloud native software. (n.d.). Retrieved June 5, 2021, from https://www.cncf.io/

Ermey, R. (2019, January 31). *The Magic Of Moving Averages*. Retrieved June 10, 2021, from https://finance.yahoo.com/news/magic-moving-averages-173300478.html

G, K. (n.d.). *N-BEATS: NEURAL BASIS EXPANSION ANALYSIS FOR INTERPRETABLE TIME SERIES FORECASTING*. Retrieved June 11, 2021, from https://kshavg.medium.com/n-beats-neural-basis-expansion-analysis-for-interpretable-time-series-forecasting-91e94c830393

Google. (n.d.). *Google Model Search*. Retrieved June 2, 2021, from https://github.com/google/model_search

Google. (n.d.). *MLOps: Continuous delivery and automation pipelines in machine learning.* Retrieved June 6, 2021, from https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning

Hanin, B., & Sellke, M. (2018). *Approximating Continuous Functions by ReLU Nets of Minimal Width.* Retrieved from https://arxiv.org/abs/1710.11278

Hen, H., & Karlesson, N. (2019, July 10). *Identification of Seasonality in Internet Traffic to Support Control of Online Advertising.* Retrieved June 10, 2021, from https://research.yahoo.com/publications/9113/identification-seasonality-internet-traffic-support-control-online-advertising

Hyndman, R. J. (n.d.). *Piecewise linear trends.* Retrieved June 10, 2021, from https://robjhyndman.com/hyndsight/piecewise-linear-trends/

Hyndman, R. J., & Athanasopoulos, G. (n.d.). *Forecasting: Principles and Practice* . Retrieved June 10, 2021, from https://otexts.com/fpp2/

IETF. (n.d.). *Operations and Management Area Working Group (opsawg).* Retrieved 07 13, 2021, from https://datatracker.ietf.org/wg/opsawg/documents/

Jenkins: build great things at any scale. (n.d.). Retrieved from https://www.jenkins.io/

Jupyter: Project Jupyter exists to develop open-source software, open-standards, and services for interactive computing across dozens of programming languages. (n.d.). Retrieved June 6, 2021, from https://jupyter.org/

Kubeflow. (n.d.). *Introduction to Katib*. Retrieved from https://www.kubeflow.org/docs/components/katib/overview/

Kubeflow: The Machine Learning Toolkit for Kubernetes. (n.d.). Retrieved June 5, 2021, from https://www.kubeflow.org/

Kubernetes: Production-Grade Container Orchestration. (n.d.). Retrieved 06 05, 2021, from https://kubernetes.io/

MinIO: Object Storage for the Era of the Hybrid Cloud. (n.d.). Retrieved June 5, 2021, from https://min.io/

Network monitoring. (n.d.). Retrieved 07 06, 2021, from https://en.wikipedia.org/wiki/Network_monitoring

Open Neural Network Exchange. (n.d.). Retrieved from https://onnx.ai/

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., . . . DeVito, Z. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems 32*, (pp. 8024--8035).

Prophet: Forecasting at Scale. (n.d.). Retrieved from https://facebook.github.io/prophet/

PyTorch: TORCHSERVE. (n.d.). Retrieved June 6, 2021, from https://pytorch.org/serve/

Quittek, J., Zseby, T., Claise, B., & Zander, S. (2004). *Requirements for IP Flow Information Export (IPFIX).* Retrieved from https://www.rfc-editor.org/info/rfc3917

Roughan, M. (n.d.). *Internet Traffic Matrices.* Retrieved 07 16, 2021, from https://roughan.info/project/traffic_matrix/

Santos, O. (2016). *Network Security with NetFlow and IPFIX: Big Data Analytics for Information Security.* Cisco Press.

SciKit Learn. (n.d.). *Feature Selection*. Retrieved from https://scikit-learn.org/stable/modules/feature_selection.html

Tensorflow: Serving Models. (n.d.). Retrieved June 6, 2021, from https://www.tensorflow.org/tfx/guide/serving

The M4 Competition Team. (n.d.). *M4 Competition: Updates*. Retrieved from https://forecasters.org/blog/2018/01/19/m4-competition/

Time-series compression algorithms, explained. (n.d.). Retrieved 07 22, 2021, from https://blog.timescale.com/blog/time-series-compression-algorithms-explained/

Wikipedia. (n.d.). *Ablation (artificial intelligence)*. Retrieved June 6, 2021, from https://en.wikipedia.org/wiki/Ablation_(artificial_intelligence)

Wikipedia. (n.d.). *Autoencoder.* Retrieved from https://en.wikipedia.org/wiki/Autoencoder

Wikipedia. (n.d.). *Automatic Differentiation*. Retrieved June 5, 2021, from https://en.wikipedia.org/wiki/Automatic_differentiation

Wikipedia. (n.d.). *Autoregressive integrated moving average*. Retrieved June 10, 2021, from https://en.wikipedia.org/wiki/Autoregressive_integrated_moving_average

Wikipedia. (n.d.). *Backpropagation*. Retrieved June 2, 2021, from https://en.wikipedia.org/wiki/Backpropagation

Wikipedia. (n.d.). *Benchmarking Methodology for Network Interconnect Devices*. Retrieved 07 09, 2021, from https://datatracker.ietf.org/doc/html/rfc2544

Wikipedia. (n.d.). *Carl Friedrich Gauss*. Retrieved June 2, 2021, from https://en.wikipedia.org/wiki/Carl_Friedrich_Gauss

Wikipedia. (n.d.). *Command-line interface*. Retrieved 07 06, 2021, from https://en.wikipedia.org/wiki/Command-line_interface

Wikipedia. (n.d.). *Curse of Dimensionality*. Retrieved June 2, 2021, from https://en.wikipedia.org/wiki/Curse_of_dimensionality

Wikipedia. (n.d.). *Data compression ratio.* Retrieved 07 21, 2021, from https://en.wikipedia.org/wiki/Data_compression_ratio

Wikipedia. (n.d.). *DevOps*. Retrieved from https://en.wikipedia.org/wiki/DevOps

Wikipedia. (n.d.). *Exponential Growth*. Retrieved from https://en.wikipedia.org/wiki/Exponential_growth

Wikipedia. (n.d.). *Graphics processing unit*. Retrieved June 2, 2021, from https://en.wikipedia.org/wiki/Graphics_processing_unit

Wikipedia. (n.d.). *Internet Control Message Protocol*. Retrieved 08 09, 2021, from https://en.wikipedia.org/wiki/Internet_Control_Message_Protocol

Wikipedia. (n.d.). *IP Flow Information Export.* Retrieved 07 13, 2021, from https://en.wikipedia.org/wiki/IP_Flow_Information_Export

Wikipedia. (n.d.). *Iperf*. Retrieved 07 09, 2021, from https://en.wikipedia.org/wiki/Iperf

Wikipedia. (n.d.). *Isaac Newton*. Retrieved June 2, 2021, from https://en.wikipedia.org/wiki/Isaac_Newton

Wikipedia. (n.d.). *IS-IS*. Retrieved 07 06, 2021, from https://en.wikipedia.org/wiki/IS-IS

Wikipedia. (n.d.). *Matrix Multiplication*. Retrieved June 2, 2021, from https://en.wikipedia.org/wiki/Matrix_multiplication

Wikipedia. (n.d.). *Microservices*. Retrieved from https://en.wikipedia.org/wiki/Microservices

Wikipedia. (n.d.). *MTR (software)*. Retrieved 07 09, 2021 , from https://en.wikipedia.org/wiki/MTR_(software)

Wikipedia. (n.d.). *NETCONF*. Retrieved 07 06, 2021, from https://en.wikipedia.org/wiki/NETCONF

Wikipedia. (n.d.). *Network Architecture Search*. Retrieved from https://en.wikipedia.org/wiki/Neural_architecture_search

Wikipedia. (n.d.). *Nyquist frequency.* Retrieved 07 16, 2021, from https://en.wikipedia.org/wiki/Nyquist_frequency

Wikipedia. (n.d.). *Open Shortest Path First*. Retrieved 07 06, 2021, from https://en.wikipedia.org/wiki/Open_Shortest_Path_First

Wikipedia. (n.d.). *Recurrent Neural Network*. Retrieved June 11, 2021, from https://en.wikipedia.org/wiki/Recurrent_neural_network

Wikipedia. (n.d.). *Reinforcement Learning*. Retrieved from https://en.wikipedia.org/wiki/Transfer_learning

Wikipedia. (n.d.). *Representational state transfer*. Retrieved from https://en.wikipedia.org/wiki/Representational_state_transfer

Wikipedia. (n.d.). *Residual Neural Network*. Retrieved June 11, 2021, from https://en.wikipedia.org/wiki/Residual_neural_network

Wikipedia. (n.d.). *Service Mesh*. Retrieved from https://en.wikipedia.org/wiki/Service_mesh

Wikipedia. (n.d.). *Simple Network Management Protocol*. Retrieved 07 06, 2021, from https://en.wikipedia.org/wiki/Simple_Network_Management_Protocol

Wikipedia. (n.d.). *Software as a service*. Retrieved from https://en.wikipedia.org/wiki/Software_as_a_service

Wikipedia. (n.d.). *Structured programming*. Retrieved June 2, 2021, from https://en.wikipedia.org/wiki/Structured_programming

Wikipedia. (n.d.). *Tensor Processing Unit*. Retrieved June 2, 2021, from https://en.wikipedia.org/wiki/Tensor_Processing_Unit

Wikipedia. (n.d.). *Transfer Learning*. Retrieved from https://en.wikipedia.org/wiki/Transfer_learning

Wikipedia. (n.d.). *YANG*. Retrieved 07 06, 2021, from https://en.wikipedia.org/wiki/YANG

Zhang, D., Mishra, S., Brynjolfsson, E., Etchemendy, J., Ganguli, D., Grosz, B., . . . Perrault, R. (n.d.). *The AI Index 2021 Annual Report.* Retrieved from arXiv: https://arxiv.org/abs/2103.06312

