Convolutional Neural Net for Time Series

Yinan Liu
William & Mary

1 INTRODUCTION

In a typical time series problem, we are given a time-indexed sequence $x_1, x_2, \ldots, x_T$ that is either vector- or real-valued, and our goal is to forecast x_{T+1}. This formulation template finds a wide range of applications in areas such as power management, public health, and finance. These problems usually possess high-frequency and low signal-to-noise properties [6, 9, 14]. High-frequency implies that the dataset is sufficiently large so that we may build a complex model from the data. Low signal-to-noise refers to that a large fraction of the response is pure noise that cannot be predicted.

Recent research has demonstrated the power of machine learning tools for time-series problems. Both standard machine learning tools and new specialized tools have been found to deliver impressive performance across diverse types of data [9, 11, 13, 14]. Perhaps a most profound discovery is that deep architecture, especially convolutional neural nets, could be effective. Therefore, increasing effort is committed to optimize the performance of CNN for time series. Nevertheless, this line of research is severely constrained by the lack of computing infrastructural support, so usually is highly ad-hoc. Researchers excessively rely only on off-the-shelf machine learning tools and use them "as it is". For example, while pytorch and tensorflow [1, 12] have proven to be effective in supporting large-scale ML research in Google/Facebook, most organizations that lack luxurious resources perform modeling research by using jupyter notebook, tweaking hyper-parameters, and occasionally use a GPU to train a model. Few productivity tools are available to scale up the research process because in part that time series learning is newer than other areas.

This work aims to address the productivity issue. Our high-level and ultimate question is:

How modern HPC, compiler, and scheduling techniques can be applied to speed up the training process?

One important point that has been often overlooked by the CS/HPC community is that the training is a long and ad-hoc process, and is much more than executing a well-defined process in GPU. Researching and training a model requires one to perform careful feature engineering, and massively test out different blackboxes including deep learning with different architectures. Cutting down the wall-clock time for researching/training models is both a usability problem (how the API can be set up so that ideas can be quickly turned into binaries), a scheduler problem (how to coordinate distributed resource for jobs that require moderate compute power), and a HPC/compiler problem (how specialized techniques are needed to speed up neural nets).

Our question and work. We next elaborate the specific questions we investigate.

(1) **Understanding the typical workflow and workload.** Designing scheduling or HPC techniques usually requires an accurate characterization of "typical workloads" for training time-series models. The research barrier here is exactly that no typical workload is available. Standard static benchmarking approaches (i.e., creation of a few datasets) are not suitable because modeling research is highly ad-hoc, and usually results in highly dynamic workloads. We rarely know the optimal model upfront so more often than not we do not know what specific architecture needs to be optimized. Therefore, we aim to prototype a *stylized but realistic research framework* that mimics what a professional time-series researcher would use. We then use the research framework for both computational and statistical purposes. *(i) Statistical purpose:* we aim to understand the efficacy of CNN-based models for a specific finance dataset. Confirming the efficacy of CNN-based models (i.e., they are "promising") would allow us to continue using the framework to produce realistic workloads. *(ii) Computation purpose.* We would like to characterize the structure of training jobs and understand how robust scheduling algorithms can be designed.

(2) **Fine-tune an important deep learning model.** We also will address a specific but commonly-seen data and computation duplication problems when deep architectures are used to learn time-series models.

Therefore, compared to most existing HPC research for speeding up a specific model, the work here focuses more on identifying all pain points that could potentially occur in the modeling process and tackles a specific data redundancy problem that arises when a deep learning architecture is used to learn time series.

2 PRELIMINARY

Background and notation. We first examine the general paradigm for building machine learning models for time-series analysis. Let $\{x_i\}_{0 \le i \le T}$ be a time series under consideration. Our goal is to forecast the time series x_{t+1} in every round based on the information available up to time t.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In machine learning's language, x_{t+1} is the prediction target (response), whereas we may use features constructed from $\{x_i\}_{i\le t}$. Recent years of research [4, 9, 11, 13, 14] suggests that time series forecast is not different from standard machine learning problems: it is crucial to build effective features from historical information, and find suitable blackbox to find the functional relationship between features.

Existing feature engineering and blackboxes. (i) Features. Features are usually manually built through domain expertise [4, 13]. For time series sensor data collected from the patients, medical experts produce the features. For financial time-series, features are usually derived from the so-called technical factors, which are constructed by the recent price and volume movement of an asset. The underlying "theory" is that trading activities in an exchange provide sufficient information about the short term future movement of the prices. *(ii) Machine learning tools/blackboxes.* It has been observed either standard ML tools or specialized ones are effective in learning relationship between features and signals [7].

Effort of using convolutional neural nets. It is natural to interpret the time-series as a one-dimensional image and use convolutional neural net to automatically extract features and/or local structures of the curve for the purpose of prediction. This idea was examined in the context of high-frequency trading in finance [3, 15] and other time series problems [10]. Using convolutional neural nets to forecast time-series is based on two intuitions:

(i) Pattern matching. We expect that a long time series could exhibit local patterns that may provide forecasting power for the future time-series.

(ii) Learning non-linear functions. The hypothesis here is that the future time series value could be a potentially non-linear function of the historical trajectory, i.e., we want to learn a non-linear function $x_t = f(x_{t-1}, x_{t-2}, \dots x_{t-L}) + \epsilon$, where ϵ is a noise term. Models built under this hypothesis can be viewed as a generalized of AR(L) model, which assumes the next time-series value is a linear function of the L lagging terms. Note on the other hand that artificial neural networks can provably learn any arbitrary non-linear functions so it can be used to learn $f(\cdot)$.

An illustrative example. We give an illustrative example on how a neural net with a simple ReLU activation function can be used to learn sine functions with the presence of noise. See Fig. 1. Note that any function can be transformed into the sum of different sine functions (at different frequencies). So this example gives strong evidence that neural nets can reliably learn polynomials.

Architecture of (convolutional) neural nets. Two types of neural nets are widely used for time-series predictions.

(i) Multi-layer perceptron. We assume that

$$x_t = f(x_{t-1}, x_{t-2}, \dots x_{t-L}) + \epsilon$$

and use a multi-layer percetron to represent $f(\cdot)$. The simplest multi-layer perceptront consists of liner layer, a ReLU layer, and another linear layer, and is shown to be able to express any functions [2]. Deep MLP sometimes are also used because they are believed to be easier to trained [5].

(ii) Convolutional neural nets. We may also treat a time-series as a one-dimensional image and apply convolutional layer to the "image" to identify local structures. We assume that x_t is only a function of $x_{t-1}, \dots, x_{t-L}$ so the "image" we use to predict x_t is $x_{t-1}, \dots, x_{t-L}$. A simple CNN consists of the following layers (from bottom to top): a (1d)-convolutional layer, a ReLU, a max-pooling layer, a linear layer, a ReLU, and another linear layer. Note that one may use more than one convolution or Linear-ReLU-linear layer to make the architecture deeper.

Training and test a model. We next explain in details on how training and test are used for training MLP or CNN. The input (feature) to both models is the lagging L terms. For example, when $L = 5$, the structure of the data is presented in Figure 2.

One usually uses multiple years of time-series data to train one model. We remark that we usually do not use training-validation (e.g., randomly hold-out 20% of the data to form the validation set for the purpose of tuning hyper-parameters) paradigm. Instead, one usually use the forward-validation method to train a time series model. For example, consider the problem of forecasting asset returns using past 10 years of data. One would produce three sets of test/validation data, each of which cover data from 2019, 2020, and 2021 respectively. When we need to evaluate the model performance for test set 2019, we would use the data prior to 20190101 in the training set. Similarly, to evaluate the performance of 2020, one would use data prior to 2019. In this way, the test set "slides" from left to right, and we never use any "future" data, i.e., no data after 2019 is used to train a model for 2019. Random training-validation split allows the usage of future data and often reduces the estimation accuracy of a model's performance.

Tuning hyper-parameters. An extensive architecture search usually is needed to determine the optimal model. Important hyper-parameters include: *(i) number of hidden nodes in intermediate layers, (ii) the dimension of the kernel, (ii) the number of kernels, (iv) number of layers, (v) learning rates, (vi) number of epochs used in training, and (vii) the magnitude of randomly initialized learnable parameters.*

(i)-(iv) are standard in neural architecture search. (v) and (vi) are also not uncommon for the purpose of tuning optimization procedures and achieving optimized variance-biased tradeoff. (iv) is quite unique to time-series learning. The learnable parameters are usually initialized in a way that the magnitude of the forecast is the same as that of the response [8]. But in time-series settings, predictive power of a model usually is not high so that the magnitude of the forecasts should not be the same as that of the response. Therefore, the magnitude of the random initialization of learnable parameters also needs to be learned.

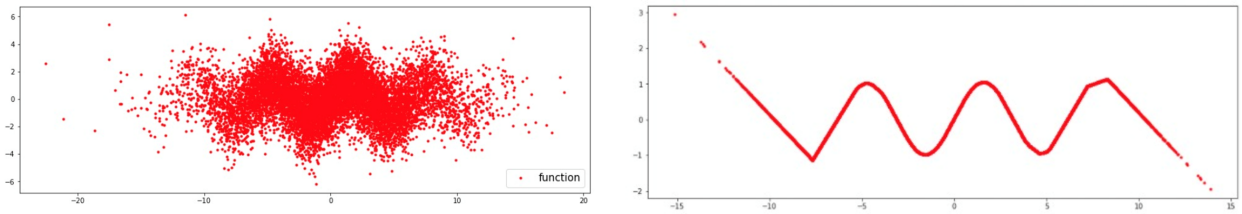

Figure 1: Learning a sine function using Pytorch. Left: the input synthetic data are generated from the model $y = \sin(x) + \epsilon$, where ϵ is a Gaussian noise. Right: when we use a multi-layer perceptron model with suitable hypara-parameters, we learn a function that is close to the ground-truth. Note that the noise is automatically removed during the training process. We also remark that any function can be decomposed as the sum of sine functions at different frequencies by using Fourier transform. So being able to learn sine functions provide strong evidence that the network can learn any polynomial.

Obs	y	f 1	f 2	f 3	f 4	f 5
1	x_6	x_1	x_2	x_3	x_4	x_5
2	x_7	x_2	x_3	x_4	x_5	x_6
...						
n	x_k	x_{k-5}	x_{k-4}	x_{k-3}	x_{k-2}	x_{k-1}

Figure 2: Transforming a time series $x_1, x_2, \ldots, x_T$ into observations for a model that uses 5 lagging terms, each of which corresponds to a feature. For example, in the first observation, x_6 is the response (the "y") and $x_1, \ldots x_5$ are the features.

Characterization of workloads and redundancies. We have the following observations about the training process. *(i) Small but massive amount of training.* Neither the training data nor the CNN/MLP model is too large. For example, when we use GPU to train and move all the training data to the memory upfront, approximately 3G of memory will be used. However, the hyper-parameter search space is large so a large number of hyper-parameters need to be trained. *(ii) Redundant data representation.* We also notice that when the data is represented in standard format (Fig.2), they exhibit significant redundant problems, e.g., the first two rows highly overlap. Observe also that when a convolutional neural net is applied to the time series, significant redundancy in computation also occurs. Therefore, while using standard data format enables us to use off-the-shelf blackboxes in a straightforward manner, this format result in inefficient storage and computation usage.

2.1 Outline of our work.

This section explains the work we have performed for the course project, and summarizes incomplete works (those that were originally planned but cannot be finished).

(1) *A realistic scheduler.* We design a scheduling system that is friendly to time series researchers. The scheduling system is optimized for fine-tuning hyper-parameters and serves for two purposes: (i) the scheduler will actually be used. The scheduler is designed in a way that new machine learning research that fully utilize William & Mary's computing environment can be done, and (ii) the scheduler generates realistic job loads. Because the scheduler will actually be used, the job loads generated by the scheduler will be realistic.

(2) *A convolution trick to eliminate data redundancy.* We introduce a so-called convolution trick to remove the data redundancies. This trick can ensure that no single data point will be replicated so it improves the storage usage efficiency up to a factor of L, where L is the number of features.

Other future works. Below are items that appear in the proposal but we did not have chance/time to implement them. *(i) Extensive evaluation:* while we extensively used the job scheduler to train massive models, we did not have a chance to thoroughly evaluate thes statistical behavior of the models. *(ii) Test the efficacy of convolutional tricks.* Convolutional tricks are only correctly implemented for a few major models. The efficacy of this solution is not extensively tested.

3 SYSTEM ARCHITECTURE AND JOB SCHEDULER

Design principles. We aim to design a system that finds a sweet-spot between the researcher's productivity and effective utilization of large computing infrastructure. Researchers' productivity refers to that researchers can effectively compile ideas (different data manipulations such as how NaN shall be handled and different architecture) into compute jobs, whereas utilization of large computing infrastructure refers to that the jobs can scale to use resources available in a typical university. This implies at the user interface front, we need a highly streamlined pipeline that allows the users to configure models, whereas at the job scheduling front, the system needs to integrate well with public computing resources.

Overview of the architecture. See Fig. 3 for the overview of our architecture. The architecture consists of two components, including the creation of jobs (data and model module in Fig. 3), and job schedulers.

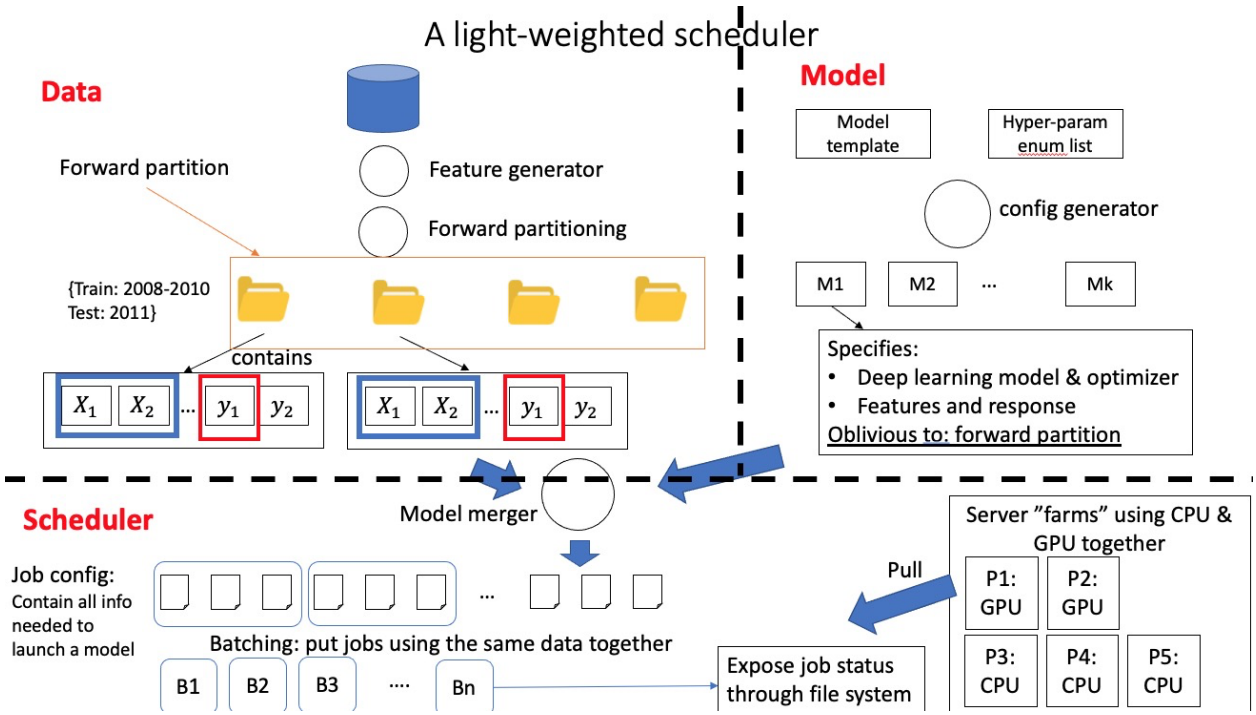

Figure 3: Overview of the architecture. The system consists of three major modules. The data module governs how the data shall be partitioned into blocks, the modeling module specifies the collection of neural networks we aim to train, and the scheduler is a system that put the lined-up jobs into different servers.

Concrete jobs are created from the output of the data and model modules. Each specific architecture and dataset will result in a unique job, which is automatically handled by a script called module merger. Each job (configuration file) should consist of sufficient information (where is the model and where is the data) to execute the training. It also needs to provide information for the scheduler to determine the order and place of execution.

Scheduling. In a typical computing environment in a university, we usually have access to multiple servers; NFS is usually available (but slow) whereas root-access is not available. Therefore, it is unlikely we can use off-the-shelf schedulers (installation and tweaking the configurations all require root access). Our scheduler is designed under these constraints. First, the scheduler will group jobs that use the same data source together so that reloading of data can be avoided when we switch between jobs. Second, multiple processes (that could possibly reside in different servers) will pull jobs through NFS. Specifically, files related to the states of the jobs will be stored and updated in a directory that can be seen by all the servers. Each process reads the configuration file and determines the next job to execute. The training results will also be stored in a directory that can be accessed by all servers.

Creation of jobs. A specific job is determined by *(i) the architecture of the neural net*, and *(ii) the training and test data.*

Hyper-parameter search usually influences the architecture of the neural net (e.g., the depth and width of the network), whereas the data manipulation, such as how training and test should be determined, or how the outliers should be processed, usually is orthogonal to the neural architecture. Therefore, the model module (controlling hyper-parameter search) and the data module (controlling generation of training and test data) shall be decoupled.

The data module outputs files with rigid format (e.g., it will contain k directories for k test sets and for each directory, the responses are all in a file called Y, whereas the features are all in a file called X). The model module allows one to specify the search space of the hyper-parameter and uses a script to translate each hyper-parameter set to an architecture.

4 OVERHEAD IN BUILDING TIME-SERIES CNN

As discussed (e.g., Fig. 2), when the historical time-series is used as an input to a convolutional neural net in the straightforward way, we see significant information and computation redundancy *(i) Storage redundancy:* Observe that for example, features between two consecutive observations overlap significantly. *(ii) Computation redundancy:* Observe also that when we apply convolutional layers to the same time-series, most convolution operations are also redundant.

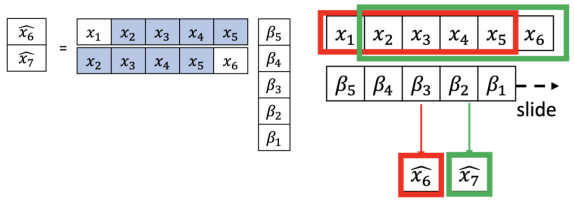

Figure 4: AR(5) in the standard form vs using the convolutional layer to mimic the computation.

(Ab)use the convolutional layer to remove redundancy. We shall explain that a careful way of using convolutional neural net layers enable us to remove information and computation redundancy. We shall start with explaining the mechanics of our solution for linear models. We then proceed to explain how this layer can be generalized to other types of layers. Finally we will explain the limitations associated with this approach.

1. Convolution Layer for Linear regression. Recall that an auto-regressive model with L lag terms assume that the next-period time-series data is a linear function of the past L data points:

$$x_t = \beta_1 x_{t-1} + \beta_2 x_{t-2} + \ldots + \beta_L x_{t-L} + \epsilon. \tag{1}$$

When a data matrix is assembled for a standard module for fitting linear regressions, the data matrix already exhibits a high degree of redundancy. So statisticians have been suffering from this problem.

Our crucial observation is that we may utilize convolution layer to mimic the computation for the linear models. See Fig. 4 Left of Fig. 4 uses the standard matrix representation to produce the forecast for x_6 and x_7. Right of Fig. 4 shows how we can use one single convolution operation to produce multiple forecasts simultaneously for the case where $L = 5$. Here, we unroll the time-series as a one-dimensional input, and apply a length-L convolutional plate to the input. As the convolutional plate "slides" towards the right, we will obtain the desired forecast sequentially. One can see that running gradient descents on these observations $\hat{x}_6$ and $\hat{x}_7$ for both the standard linear model and convolutional model will result in identical updates. The resulting code is quite straightforward. See Listing. 1 for the architecture and Listing. 2 for the training.

2. Generalization of convolutional tricks. We next explain how the idea can be generalized to other layers and multiple layers.

2a. ReLU. Because ReLU is an element-wise operation, no special manipulation is needed.

Example: Multi-Layer Perceptron. We next explain how multi-layer perceptron (MLP) can be implemented using this idea. Recall that a MLP consists of a fully connected linear layer, followed by a ReLU, and followed by another fully connected linear layer. See Fig. 5 for the original MLP with three observations and Fig. 6 for the convolutional-trick based implementation.

We next explain how the convolutional trick can be specifically used here. Compared to the standard MLP, we use a full time series so that no data is replicated. We first use a convolutional layer to mimic the lower linear layer. In the specific example in which $L = 5$, the size of the kernel is set to be 5. The number of kernels should be the same as the number of hidden nodes w. Then we apply an element-wise ReLU. The dimension of the output will be $3 \times w$, where 3 comes from $7 - 5 + 1$. Then to mimic the top linear layer, we need to use a convolution in a different shape, i.e., the convolution layer this time has kernel size $w \times 1$ and we need only one convolution plate. At the end, the architecture outputs the forecasts for x_6, x_7, and x_8.

```python
#assume that ts is the time series
x = torch.tensor(np.float32(ts[:-1]))
x = Variable(x, requires_grad=True)

y = torch.tensor(np.float32(ts[m_len:])).T
y = Variable(y, requires_grad=True)

class Lambda(nn.Module):
    def __init__(self, func):
        super().__init__()
        self.func = func

    def forward(self, x):
        return self.func(x)


def preprocess(x):
    return x.view(-1, 1, total_obs-1)

model = nn.Sequential(
    Lambda(preprocess),
    nn.Conv1d(1, 1, m_len),
    Lambda(lambda x: x.view(x.size(0), -1)),
)
```

Listing 1: Using convolutional neural nets to mimic AR(L) models.

```python
def loss(input, target):
    return torch.norm(input-target)

loss_func = nn.MSELoss()

lr = 0.01
epochs = 100
bs = 200
n = X.shape[0]

opt = optim.SGD(model.parameters(), lr=lr, momentum=0.0)

for epoch in range(epochs):
    xb = x
    yb = y
    model.train()
    predy = model(xb)
    loss = loss_func(predy, yb)
    loss.backward()
    opt.step()
    opt.zero_grad()
    if epoch % 10 != 0:
        continue

    with torch.no_grad():
        cov = np.corrcoef(np.squeeze(predy.cpu().detach().
              numpy()), np.squeeze(yb))
        print('epoch', epoch, ': corr coef = ', cov[0, 1])
```

Listing 2: The training algorithm.

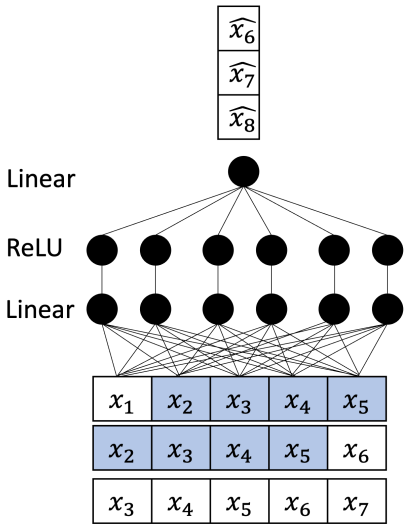

Figure 5: A standard MLP. In this example, the dimension of the input is 5. The first layer is a linear layer. Let w be the number of hidden nodes. The next layer is ReLU. The final layer is another linear layer.

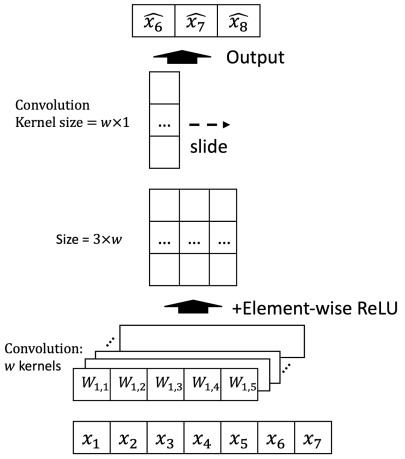

Figure 6: Compiling a standard MLP (shown in Fig. 5) into a convolutional-trick based model.

2b. Max-pooling. Max-pooling requires the most complex operations. See Fig. 7 for the max-pooling operation for four observations. Here, we assume that we examine $L = 6$ lags and the kernel size of the max pooling is 3. We assume the default stride parameter (i.e., stride size equals kernel size). Let us walk through an example. The original time series is $x_1, x_2, x_3, x_4, x_5, x_6$. After the max-pooling, we obtain $v_1 = (\max\{x_1, x_2, x_3\}, \max\{x_4, x_5, x_6\}$. The max-pooling for the subsequent observations can be computed in a similar manner. Fig. 8 gives an example of four observations. Let us refer to the outcome after the max-pooling for these four observations as v_1, v_2, v_3, and v_4 respectively.

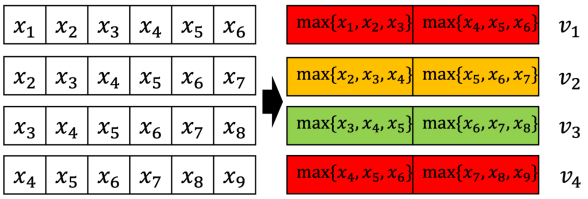

Figure 7: Using a standard max-pooling with kernel size being 3 for four consecutive observations. Each observation consists of 6 lagging terms.

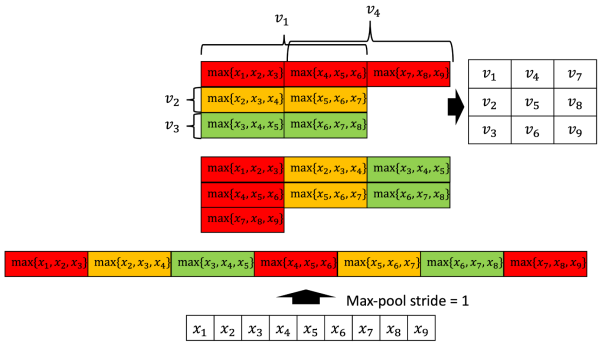

Figure 8: Workflow of using the convolution trick to compile max-pooling, which needs to involve reshape and transpose operations

We next explain how the max-pooling operation can be implemented using the convolutional trick. First, we observe that we need to set the stride of the max pooling to be 1 (instead of the default value). Otherwise, the max-pooling layer would skip the computation of, for example, $\max\{x_2, x_3, x_4\}$. After we run a max-pooling with stride one, we can see that the resultant contains all the needed information but the information is not arranged in a "proper" way. For example, v_1 consists of two red blocks that are not consecutive to each other. We next need two operations to suitably "rotate" the result from max-pooling. First, we perform a reshape to allow all blocks in the same color being in the same column. Then we need to perform a transpose operation between the two axes. So at the end, we can properly extract $v_1, \ldots, v_9$ again.

Example: Convolutional neural nets. We may compose a convolutional neural net using the above building block. The specific convolution neural net we build consists of (from bottom to top) a convolutional layer, a ReLU layer, a max-pooling layer, a linear layer, a ReLU layer, and a linear layer. See below for the code in Listing 3.

Open questions. It appears that assembly of the whole network requires a highly manual effort. For example, let us revisit the MLP discussed above. While the two linear layers in MLP seem to serve for identical purpose, we need convolutional kernels of different shapes to "emulate" two linear layers. The first convolution layers (for

```
model = nn.Sequential(
    Lambda(preprocess),
    nn.Conv2d(1, hidden_num, (1, kernel_size)),
    nn.MaxPool2d((1, kernel_size), stride=1),
    Lambda(lambda x: x.view(x.size(0), hidden_num, -1,
          kernel_size)),
    Lambda(lambda x: torch.transpose(x, 2, 3)),
    nn.ReLU(),
    nn.Conv2d(hidden_num, hidden_num, (1, kk)),
    Lambda(lambda x: torch.transpose(x, 2, 3)),
    Lambda(lambda x: x.reshape(x.size(0), hidden_num, 1, -1))
          ,
    Lambda(lambda x: x.view(x.size(0), 1, hidden_num, -1)),
    nn.ReLU(),
    nn.Conv2d(1, 1, (hidden_num, 1)),
    Lambda(lambda x: x.view(x.size(0), -1))
)
```

Listing 3: Example code

emulating the bottom linear layer) consists of a collection of row kernels whereas the second convolution layer consists of a single column kernels. It appears that more careful work is needed to enable easy assembly of different layers.

Other concerns. Recall that in a standard stochastic gradient descent algorithm, one would need to randomly permute all the training data before running the SGD. We remark that the convolutional tricks cannot replicate the random permutation operation. This is because forecasts for data points from consecutive timesteps will need to be produced together to remove redundancy, whereas random permutation requires that forecasts/observations from consecutive timesteps not placed in the same batch.

Some of our preliminary experiments suggest that SGD and SGD with random permutation results in models with different forecasting power. So the inability to mimic random permutation remains a major drawback of the proposed solution.

Analysis. One can check the correctness of the compilation in a straightforward manner. The size of training data will be proportional to the length of the time series, which is a factor of L better than the standard data format. In practice L is usually in the order of 10's for MLP and 100's for CNN.

5 EXPERIMENTS AND OBSERVATIONS

We checked the correctness of the convolution trick using a small number of examples. Here, we report experiments related to the scheduler. We focus on validating that *(i)* the scheduler can indeed scale up to perform massive training, and *(ii)* the training is indeed important.

Datasets. We use equity datasets from the US markets between 2010 and 2018. We focus on the top 3000 most heavily traded stocks, and the time series of their returns. There are three test datasets, using data from 2016, 2017, and 2018, respectively. Forward validation is used.

Computing environment. We use bg machines to run the jobs. Bg1 to bg5 can perform CPU-based training, whereas GPUs in BG1 is also utilized.

Hyper-parameters. We tested a total number of approximately 70 deep learning models. We also tested the impact of training length to the test performance (e.g., whether early dropout works). Finally, we also tested two batch sizes (256).

Result. The experiments ran continuously for 3 days in multiple BG machines. So we confirm that the scheduler is robust. In Fig. 9, we plot the out-of-sample correlation (larger better) for different models using different epochs. Specifically, x-axis is the number of epochs (one epoch passes the whole training data once). Dots in the same color represent the performance of the same architecture using different epochs. Two curves having the same color represents result for different test size. Each epch covers approximately 1 million training observations. In this specific dataset, when the correlation is above 0.015, it is considered as useful.

We have the following major observations: *(i)* Careful architecture search gives us statistically significant models. We observe that a substantial portion of the dots are above 0.015 but many are below. Therefore, it confirms that hyper-parameter search (e.g., a high-througput system) is crucial, *(ii)* Larger batch size is worse. We notice that the performance is substantially improved when the batch size is 256. On the other hand, GPU do not have superior performance when the batch size is small. Thus, it confirms that it is challenging to fully utilize GPU to train time-series models, *(iii) Parallel jobs are useless.* As discussed earlier, we could have execute multiple training at CPU to utilize multiple cores. But it seems that modern BLAS is effective in "sucking up all resources" so parallel execution of CPU jobs do not improve throughput.

6 CONCLUSION

This work examines how we can design schedulers and convolutional based tricks to effectively train neural net models for time series. At the scheduler front, we designed a robust high-throughput system that produces realistic workloads. The system also identifies a significant number of statistically significant models. At the CNN trick front, our trick can solve a common data redundancy problem widely appeared in time series models. The trick reduces the memory complexity to linear of the training data, which of a factor of L better than standard data format. It remains an open problem whether using this trick at massive scale can also enable us to find statistical significant models.

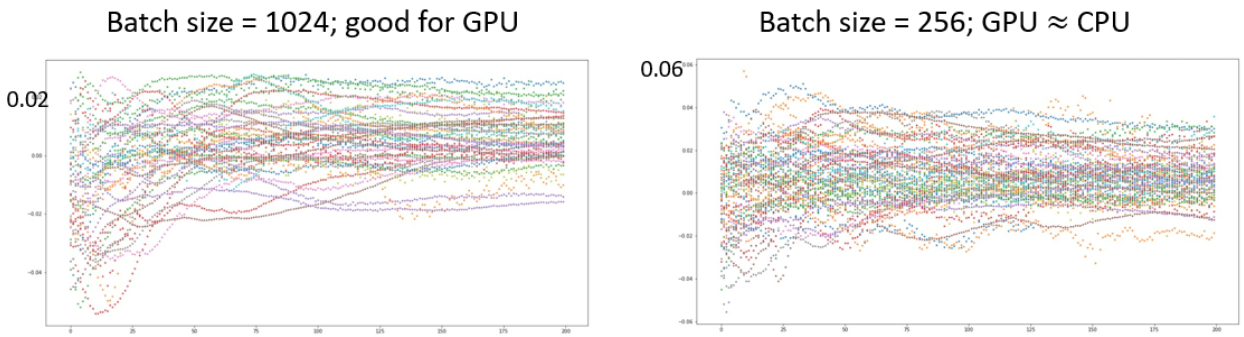

Figure 9: Experimental results for massive architecture search for different batch sizes. The batch size for the models in the left is 1024 and the batch size for the models in the right is 256.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.

[2] Alexandr Andoni, Rina Panigrahy, Gregory Valiant, and Li Zhang. 2014. Learning polynomials with neural networks. In *International conference on machine learning*. PMLR, 1908–1916.

[3] Jou-Fan Chen, Wei-Lun Chen, Chun-Ping Huang, Szu-Hao Huang, and An-Pin Chen. 2016. Financial time-series data analysis using deep convolutional neural networks. In *2016 7th International conference on cloud computing and big data (CCBD)*. IEEE, 87–92.

[4] Luyang Chen, Markus Pelger, and Jason Zhu. 2020. Deep learning in asset pricing. *Available at SSRN 3350138* (2020).

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

[6] James Douglas Hamilton. 2020. *Time series analysis*. Princeton university press.

[7] Ziniu Hu, Weiqing Liu, Jiang Bian, Xuanzhe Liu, and Tie-Yan Liu. 2018. Listening to chaotic whispers: A deep learning framework for news-oriented stock trend prediction. In *Proceedings of the eleventh ACM international conference on web search and data mining*. 261–269.

[8] Siddharth Krishna Kumar. 2017. On weight initialization in deep neural networks. *arXiv preprint arXiv:1704.08863* (2017).

[9] Chien-Liang Liu, Wen-Hoar Hsaio, and Yao-Chung Tu. 2018. Time series classification with multivariate convolutional neural network. *IEEE Transactions on Industrial Electronics* 66, 6 (2018), 4788–4797.

[10] Yongsen Ma, Sheheryar Arshad, Swetha Muniraju, Eric Torkildson, Enrico Rantala, Klaus Doppler, and Gang Zhou. 2021. Location-and Person-Independent Activity Recognition with WiFi, Deep Neural Networks, and Reinforcement Learning. *ACM Transactions on Internet of Things* 2, 1 (2021), 1–25.

[11] Youngjin Park, Deokjun Eom, Byoungki Seo, and Jaesik Choi. 2020. Improved Predictive Deep Temporal Neural Networks with Trend Filtering. *arXiv preprint arXiv:2010.08234* (2020).

[12] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019), 8026–8037.

[13] Shuangquan Wang, Gang Zhou, Jiexiong Guan, Yongsen Ma, Zhenming Liu, Bin Ren, Hongyang Zhao, Amanda Watson, and Woosub Jung. 2021. Inferring food types through sensing and characterizing mastication dynamics. *Smart Health* 20 (2021), 100191.

[14] Bendong Zhao, Huanzhang Lu, Shangfeng Chen, Junliang Liu, and Dongya Wu. 2017. Convolutional neural networks for time series classification. *Journal of Systems Engineering and Electronics* 28, 1 (2017), 162–169.

[15] Muhammad Zulqarnain, Rozaida Ghazali, Muhammad Ghulam Ghouse, Yana Mazwin Mohmad Hassim, and Irfan Javid. 2020. Predicting Financial Prices of Stock Market using Recurrent Convolutional Neural Networks. *International Journal of Intelligent Systems and Applications* 12, 6 (2020), 21–32.