Preparation Phase Report

Forward Propagation Through Time Hardware Acceleration for Efficient Training of Recurrent Neural Networks

Yicheng Zhang, 1720384
y.zhang1@student.tue.nl
Embedded Systems

September 1, 2023

Contents

1 Introduction

Intelligent devices targeting running deep learning algorithms at the edge, with the computation and data storage being close the the data source, are normally loaded with network models well pre-trained by GPUs. However, in the probably uncontrollable environment changes may happen where edge devices are deployed, partial retraining is required for personalization, or models need to be fine tuned.[1] Then new features should be captured and learned by the network models, while in this case, the GPU may not be an ideal solution. For one thing, communication with GPUs, retraining and re-deployment of models to edge devices can be time-consuming which puts jobs at the edge on hold. Moreover, accessing the remote GPU may not be an option sometimes because of privacy or security issues. For another, although GPU-based platforms provide high computation throughput for large mini-batch deep neural network computations, they may perform inefficiently, i.e.low utilization, when approaching a few new features which can have small batch sizes and irregular sequence lengths, especially for the recurrent neural network (RNN). Therefore, acceleration systems are needed for efficient, in-place and lightweight training at the edge.

On the other hand, the Back Propagation Through Time (BPTT) algorithm to train RNN is computation and memory-intensive, which makes it expensive on edge devices, while it can provide satisfactory generalization. Lately, a novel RNN training algorithm Forward Propagation Through Time (FPTT)[2] has been proposed based on BPTT, which lowers the computation complexity and the memory requirement of BPTT on sequence modeling tasks and terminal prediction tasks at the cost of more memory access, meanwhile the resulting accuracy is no lower than BPTT can reach. Thus, FPTT is a promising training method at the edge.

This report aims at preparations and preliminaries for creating a system-level solution that performs training efficiently by FPTT on edge devices. This final goal is expected to be realized by hardware and software co-design.

2 Problem Statement

Currently, there is a gap in the current state of research and development: the FPTT algorithm lacks efficient hardware implementations. This is due to algorithmic complexities, optimization challenges, and the early stage of the algorithm's development. Thus, in this thesis, we want to explore possible trade-offs between energy/area and propose an efficient hardware architecture for executing FPTT training of small batch sizes.

3 Related Work

Paper [3] proposes an algorithm-system co-design framework to make on-device training possible with only 256KB of memory. They solved two unique challenges faced with On-device training: (1) the quantized graphs of neural networks are hard to optimize due to low bit-precision and the lack of normalization; (2) the limited hardware resource does not allow full back-propagation. To cope with the optimization difficulty, they proposed Quantization-Aware Scaling to calibrate the gradient scales and stabilize 8-bit quantized training. To reduce the memory footprint, they proposed Sparse Update to skip the gradient computation of less important layers and sub-tensors. The algorithm innovation is implemented by a lightweight training system, Tiny Training Engine, which prunes the backward computation graph to support sparse updates and offload the runtime auto-differentiation to compile time.

The Edge TPU[4] is Google's purpose-built ASIC chip designed to run machine learning (ML) models for edge computing, meaning it is much smaller and consumes far less power compared to the TPUs hosted in Google data centers (also known as Cloud TPUs). The machine learning runtime used to execute models on the Edge TPU is based on TensorFlow Lite. The Edge TPU is only capable of accelerating forward-pass operations, which means it's primarily useful for performing inferences (although it is possible to perform lightweight transfer learning on the Edge TPU). The Edge TPU also only supports 8-bit math, meaning that for a network to be compatible with the Edge TPU, it needs to either be trained using the TensorFlow quantization-aware training technique, or since late 2019 it's also possible to use post-training quantization.

There are also a few training acceleration platforms for RNN, compensating for the GPU. A typical one is FARNN [1]. It is a hybrid acceleration platform for recurrent neural network training, combining FPGA and GPU, and it separates the tasks by identifying whether the job is efficient on GPU. If not, the job is handled by the dedicated computation unit in the FPGA. The evaluation result indicates that FARNN outperforms the GPU-only platform for RNN training by up to 4.2x with small batch sizes, long input sequences, and many RNN cells per layer.

In terms of learning algorithms on edge devices, biologically plausible ones with the Spiking Neural Network (SNN) are usually preferable because of the bionic power efficiency. However, they have limited accuracy and trainability, so it is hard for network models to learn complex tasks by them. Typical examples are Spike Driven

Synaptic Plasticity (SDSP) [5], Spike Timing Dependent Plasticity (STDP) [6] and Eligibility Propagation (E-Prop) [7].

4 FPTT Background

In this section, the FPTT algorithm is introduced, and before that, BPTT is explained because it is the basis of FPTT.

4.1 BPTT

Backpropagation (BP) is a machine learning algorithm that performs a backward pass to adjust the parameters of the model, in the direction that the actual output of the model approaches the expectation, that is, minimizing the loss function. Moreover, Backpropagation Through Time (BPTT) is used to apply backpropagation on unrolled RNNs.

A simple example of RNN can illustrate how BPTT updates the parameters of the network model. Assuming that an RNN has one input layer, one hidden layer, and one output layer, then its forward pass is formed by equations 1 and 2.

$$h_t = HiddenLayerActivation(x_t \cdot W_{ih} + h_{t-1} \cdot W_{hh}) \tag{1}$$

$$o_t = OutputLayerActivation(h_t \cdot W_{ho}) \tag{2}$$

By these rules, a node of network states is created after each time step. If we keep track of the states at all steps, a sequence of states generated by all input can be obtained as shown on the right part of the figure 1. With these states, i.e. an unfolded RNN, BPTT can be performed from the output of the last time step.

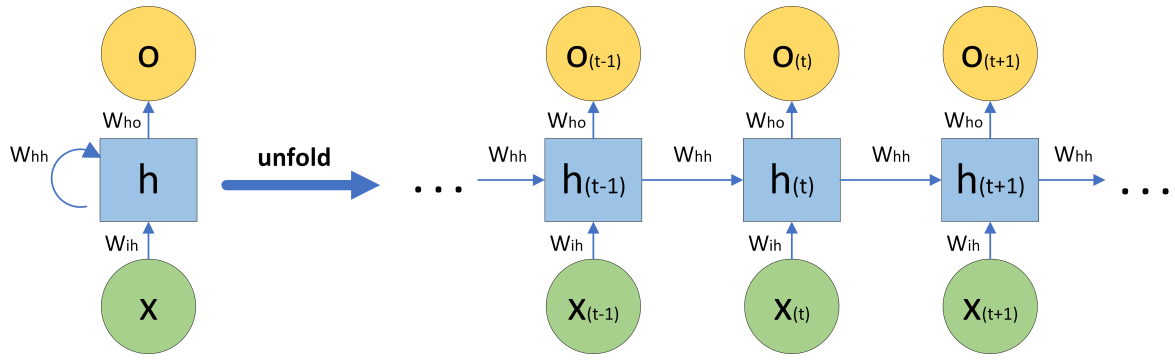

Figure 1: Topology of the basic RNN and unfolding

The way in which BPTT updates parameters can be explained by taking W_{hh} as an instance, as a condition of equation 3 and 4. Here, we assume that the task to be solved is a sequence-to-sequence problem, e.g. language modeling. In other words, the output at every time step is considered and there is one label for every element in the input sequence. Note that L is the loss function and η is the learning rate.

$$W_{hh} = W_{hh} - \eta \cdot \frac{\partial L^{(t)}}{\partial W_{hh}} \tag{3}$$

$$\frac{\partial L^{(t)}}{\partial W_{hh}} = \frac{\partial L^{(t)}}{\partial o^{(t)}} \cdot \frac{\partial o^{(t)}}{\partial h^{(t)}} \cdot \left(\sum_{k=1}^{t} \frac{\partial h^{(t)}}{\partial h^{(k)}} \cdot \frac{\partial h^{(k)}}{\partial W_{hh}}\right) = \frac{\partial L^{(t)}}{\partial o^{(t)}} \cdot \frac{\partial o^{(t)}}{\partial h^{(t)}} \cdot \left(\sum_{k=1}^{t} \left(\prod_{i=k+1}^{t} \frac{\partial h^{(i)}}{\partial h^{(i-1)}}\right) \cdot \frac{\partial h^{(k)}}{\partial W_{hh}}\right) \tag{4}$$

By alternatively iterating forward and backward passes multiple times, it is possible to find the parameter values that minimize the loss function and make the network output approach the expectation.

4.2 data-sets and handling

There are mainly two types of datasets for RNNs, i.e. sequence modeling and terminal prediction, as benchmarks for network architectures or learning algorithms. In terms of sequence modeling, namely the sequence-to-sequence task, the network receives the sequence as the input, which means the element of a sample enters the network one by one at every time step. On the other hand, the network also produces the sequence as the output. The example used in paper [2] is the language modeling task Penn Tree Bank (PTB) that expects the

sentence as the output. As that task is not considered in experiments of this report, details can be found in section 6.7 of the supplement document of paper [2], also for the following brief explanations.

With regard to terminal prediction tasks, the network also receives a sequence, but only the network output at the last time step is considered as the prediction, though there is a network output at each time step as well. The examples used are Add Task, (P)S-MNIST, and pixel-CIFAR.

Add Task has been used to evaluate long-range dependencies in RNN architectures. An example data point consists of two sequences (x_1, x_2) of the same length T and a target label y. x_1 contains real-valued entries drawn uniformly from $[0, 1]$, x_2 is a binary sequence with exactly two 1s, and the label y is the sum of the two entries in sequence x_1 where x_2 has 1s. Lengths T used are 200, 500, 750, and 1000.

(P)S-MNIST is (Permuted) Sequential MNIST or (Permuted) pixel MNIST. MNIST is a dataset of handwritten digits of the size 28x28x1 (greyscale image) for recognition. S-MNIST is fully flattened MNIST where each sample has a size of 784x1. Generally, MNIST can be handled by neural networks in three ways. First, for fully connected networks, all 784 pixels enter the network in parallel which requires the input dimension of 784. Then for recurrent networks, it is possible that one row of 28 pixels enters the network together, and it takes 28 times steps to process the entire 28 rows. To that end, the input dimension is 28, i.e. length of a row. Last, for recurrent networks, it is also feasible that only one pixel enters the network at each time step, and then 784 time steps are needed to stream the entire image. In such a way, the input dimension is 1. RNNs can have less weights to save memory space if the input dimension is smaller. The second way of processing MNIST can be handled by baseline RNNs and optimizers, while the third one is a serious test on the network about the ability to learn dependencies over a long range. LSTM can hardly approach it without an advanced optimizer like RMSprop or Adam. However, with the help of FPTT, LSTM can learn SMINST with simple SGD, optionally with Momentum. Note that PS-MNIST is the task that has permuted positions of pixels of S-MNIST, which makes S-MNIST more challenging.

pixel CIFAR is also an image classification task similar to (P)S-MNIST. The main difference is that samples are RGB images of larger size (32x32x3), so the input dimension should be 3.

4.3 BPTT online formulation

FPTT is highly related to BPTT, basically, it can be considered as a process in that a long input sequence is split into multiple sub-sequences, and parameters are updated every time BPTT is performed on a sub-sequence. In addition, an extra constraint is introduced to coordinate all updates. Therefore, the FPTT algorithm consists of two ideas, BPTT online formulation and an extra constraint.

Two problems with BPTT inspire BPTT online formulation. For one thing, BPTT requires huge memory space for long input sequences, as network states at all time steps need to be saved and then used in the calculation. On the other hand, when the length of the input sequence is longer, more elements will be involved in the production stated in equation 4. This can result in gradient vanishing or exploding, i.e., poor trainability. Hence, it is natural to consider splitting a long input sequence into smaller sub-sequences. For instance, an input sequence of 1000 steps can be broken into 10 sub-sequences of 100 steps. When the network finishes the first piece, BPTT is performed based on the network states generated by 100 steps, and the result updates the network parameters. Once the network states are used, the memory space can be released immediately and used for the next piece. In such a way, the memory space always needs to keep the network states of the last 100 steps rather than 1000 steps. Moreover, BPTT within fewer steps creates shorter production, which is more likely to avoid gradient exploding/vanishing. The parameter update rule by BPTT online formulation is given in 5. As the parameter is not time-invariant over an input sequence, $W_{hh}^{(t)}$ has one superscript t so that it represents the parameter value at time step t. It follows $0 \leq t \leq T$, while T is the sequence length and $W^{(0)}$ is the initial value.

$$W_{hh}^{(t+1)} = W_{hh}^{(t)} - \eta \cdot \frac{\partial L^{(t)}}{\partial W_{hh}^{t}} \tag{5}$$

The Add Task is a toy example that shows the effect of online formulation. As shown in the figure 2, to solve the same task traditional BPTT with one parameter update cannot decrease the loss function, while BPTT online formulation with 10 updates converges successfully.

4.4 regularization penalty

On the other hand, the regularization penalty $R(t)$ provides stability and convergence for the frequent update by BPTT online formulation, especially for harder tasks.

Basically, the online formulation treats a long input sequence by breaking it into multiple parts and performing BPTT on those parts sequentially. This can lead to the situation that a feature or dependency is separated into two or more parts, which makes it hard to be completely reflected on parameters by multiple

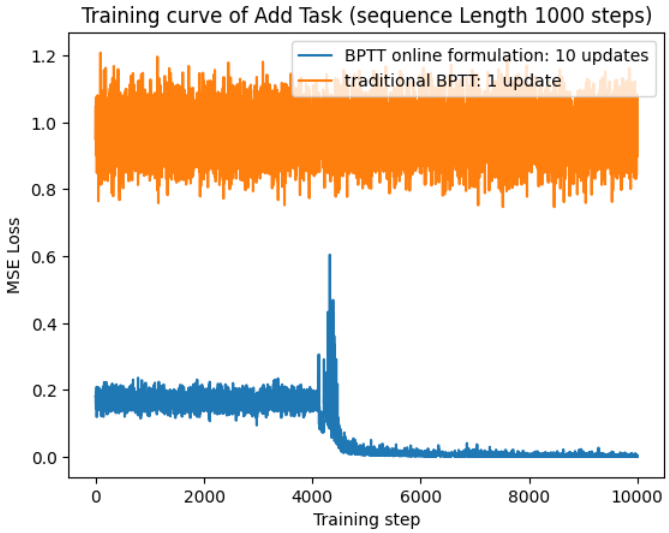

Figure 2: Training curve of Add Task (1000 time steps in sequence length)

updates, especially for tasks harder than Add Task like PTB. Moreover, frequent parameter updates may lead to instability. Therefore, an extra constraint $R^{(t)}$ is proposed and added to the loss function to correlate and stabilize the parameter updates, as shown in equations 6, 7, and 8, the weight for recurrent connections W_{hh} is still taken as an example.

$$W_{hh}^{(t+1)} = W_{hh}^{(t)} - \eta \cdot \frac{\partial (L'^{(t)} + R^{(t)})}{\partial W_{hh}^{t}} \tag{6}$$

$$R^{(t)} = \frac{\alpha}{2} \cdot \| W_{hh}^{(t)} - \bar{W}_{hh}^{(t)} - \frac{1}{2\alpha} \cdot \nabla_{l_{t-1}}(W_{hh}^{(t)}) \|^2 \tag{7}$$

$$\bar{W}_{hh}^{(t+1)} = \frac{1}{2}(\bar{W}_{hh}^{(t)} + W_{hh}^{(t+1)}) - \frac{1}{2\alpha} \cdot \nabla_{l_t}(W_{hh}^{(t+1)}) \tag{8}$$

Equation 6 is the basic rule of gradient descent, which minimizes the loss function w.r.t. the weight, with η being the learning rate. The innovation is that the loss function has an extra $R(t)$ other than the traditional loss $L'(t)$ that measures the difference between the actual network output and the expected network output.

As defined in equation 7 $R(t)$ is the penalty for regularization on the distance between the current parameters and their running average. α is a hyperparameter. In other words, with $R(t)$ being a part of the loss function to be minimized, the updated parameters are forced to be close to their running average history as much as possible. Intuitively, the parameters are changed in a conservative and consistent way, so that multiple updates resulting from BPTT online formulation can target the same prediction. It is stated in paper [2] that $\frac{1}{2\alpha} \cdot \nabla_{l_{t-1}}(W_{hh}^{(t)})$ is a correction term, as well as the $\frac{1}{2\alpha} \cdot \nabla_{l_t}(W_{hh}^{(t+1)})$, and they are the gradient of loss at the last time step w.r.t. the parameter at the next time step. Equation 8 defines the update rule for the running average of the parameter.

As the loss function is defined, we introduce the entire algorithm. First, network parameters (the weight and optionally the bias) are initialized randomly, and then the initial running averages of parameters are the same as parameters. Training usually requires multiple epochs, and batches are shuffled for each epoch and the hidden state of the network is cleared for each batch. For an application of which sequence length is T, there are T steps. In each step, the network performs forward pass and the output of the network leads to the traditional/common loss function. However, in FPTT the whole loss function also requires $R(t)$. Then the gradient of the loss function is calculated and used to update the parameter. Afterward, the running average of the parameter is updated by the new parameter. After T steps, a batch of samples is trained. The updated parameter at the end of a batch is used for the next batch. The required number of batches completes an epoch, and the required number of epochs completes the training.

4.5 mathematical foundation

Online formulation breaking the long input sequence mitigates the gradient vanishing/exploding, but it also has the unwanted side effect of breaking the feature or dependency distributed over a long sequence, particularly when there are too many sub-sequences. Then, $R^{(t)}$ is designed to solve this.

Algorithm 1 Training RNN with FPTT

Input: Training data $B = \{x^i, y^i\}_i^N$, Timesteps T
Input: Learning rate η, Hyper-parameter α, # Epoch E
Initialize: W_1 randomly in the domain W
Initialize: $\overline{W_1} = W_1$
for $e = 1$ **to** E **do**
    randomly shuffle B
    for $i = 1$ **to** B **do**
        Set: $(x, y) = (x^i, yi)$ and $h_0 = 0$
        for $t = 1$ **to** T **do**
            Update: $h_t = f(W, x_t, h_{t-1})$
            $l_t(W) = l_t(y_t, v^T h_t)$
            $l(W) = l_t(W) + \frac{\alpha}{2}||W - \overline{W}_t - \frac{1}{2\alpha}\nabla_{l_{t-1}}(W_t)||^2$
            $W_{t+1} = W_t - \eta\nabla_W l(W)|_{W=W_t}$
            $\overline{W}_{t+1} = \frac{1}{2}(\overline{W}_t + W_{t+1}) - \frac{1}{2\alpha}\nabla_{l_t}(W_{t+1})$
        end for
        Reset: $W_1 = W_T$ and $\overline{W}_1 = \overline{W}_T$
    end for
end for
Return: W_T

In Section 6.7 of this document[1], the author proves that the BPTT online formulation together with the regularization penalty can reach a W_∞ which is the stationary point of the traditional BPTT solution.

4.6 algorithm generalization

Two important generalizations are made with FPTT which make this algorithm more widely applicable.

For one thing, FPTT-K is a superclass of FPTT theory that provides a trade-off between memory usage and update frequency. K is simply the number of parameters updated by FPTT on an input sequence or the number of sub-sequences into which the input sequence is divided. Note that the default K of the FPTT theory is the length of the input sequence. If a long sequence is exactly divided into K, there will be K updates in total. For instance, if we want to apply FPTT-20 on an input sequence of 100 steps, we will have 20 sub-sequences each having 5 steps. In such a way, the memory space needs to keep the network states in 5 steps. Alternatively, when K is larger, the memory space can be smaller to keep fewer steps in the network states. When K equals the sequence length, there only needs to be one step of network states in the memory.

Especially, when the length of the sequence cannot be divided by K exactly, as when we need to apply FPTT-18 to the same 100-step sequence, we will first have 18 parts each having 5 steps (floor division, 100/18=5) and one reminder part of 10 steps (100-18*5 = 10).

Also, note that the BPTT algorithm has the terminology BPTT-K, which may be confused with FPTT-K. It is a different concept that for BPTT with truncation, K means how many earlier steps, counting backward from the current step, will be referred to the BPTT calculation.

In other words, in terms of terminal prediction applications like image classifications, one input sequence has only one label. Then, to enable FPTT on terminal predictions, multiple update labels must be created, as shown in equations 9 and 10.

$$l_t = \beta \cdot l_t^{CE} + (1 - \beta) \cdot l_t^{Div} \tag{9}$$

$$l_t^{CE} = -\sum_{\overline{y} \epsilon y} \mathbf{1}_{\overline{y}=y} log\hat{P}(\overline{y}); l_t^{Div} = -\sum_{\overline{y} \epsilon y} Q(\overline{y}) log\hat{P}(\overline{y}); \beta = \frac{t}{T} \tag{10}$$

$\hat{P}$ is the current estimate of the label distribution, so the classification loss l_t^{CE} is the loss of cross entropy between the prediction and the label. Q is estimated in the last training epoch, then auxiliary loss/oracle loss l_t^{Div} is the cross-entropy loss between the current prediction and the prediction of the last epoch. With β being the current time step over the sequence length, it follows that towards a long input sequence the prediction is first forced to be close to the history prediction to keep stable and then approaches the real label.

[1] http://proceedings.mlr.press/v139/kag21a/kag21a-supp.pdf

4.7 comparison

FPTT can be compared with other solutions in terms of cost and training results.

Paper [8] provides the comparison (table 1) FPTT(-K) between BPTT and E-Prop (a typical bio-plausible learning algorithm) with regard to three types of cost, assuming that the task is the sequence-to-sequence problem. Note that the constant associated with the gradient, computation is a monotonically increasing function $c(.)$ of the sequence length, i.e. $c(1) < c(K) < c(T)$.

Table 1: Computational cost for gradient, parameter update & memory storage overhead

Algorithm	Gradient Updates	Parameter Updates	Memory Storage
BPTT	$\Omega(c(T)T)$[1]	$\Omega(1)$	$\Omega(T)$
FPTT	$\Omega(c(1)T)$	$\Omega(T)$	$\Omega(1)$
FPTT-K	$\Omega(c(K)T)$	$\Omega(K)$	$\Omega(T/K)$
E-Prop	$\Omega(c(1)T)$	$\Omega(T)$	$\Omega(1)$

[1] on the sequence-to-sequence task

For the training result, FPTT outperforms BPTT in accuracy on many benchmarks, as shown in the table 2. Furthermore, it enables the network to learn long sequences that BPTT cannot train, as shown in figure

Table 2: Accuracy Comparison

Algorithm	S-MNIST	PS-MNIST	CIFAR-10
BPTT	97.71%	88.91%	60.11%
FPTT[1]	98.67%	94.75%	71.03%

[1] K used here is 10.

4.8 LSTM

Long short-term memory (LSTM)[9] network is a recurrent neural network (RNN). A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell. Forget gates decide what information to discard from a previous state by assigning a previous state, compared to a current input, a value between 0 and 1. A (rounded) value of 1 means to keep the information, and a value of 0 means to discard it. Input gates decide which pieces of new information to store in the current state, using the same system as forget gates. Output gates control which pieces of information in the current state to output by assigning a value from 0 to 1 to the information, considering the previous and current states. Selectively outputting relevant information from the current state allows the LSTM network to maintain useful, long-term dependencies to make predictions, both in current and future time steps.

The expressions of LSTM in vector notation (a layer of multiple cells) are given from equation 11 to 16. Variables in lowercase are vectors while the ones in uppercase are matrices. $W_{\star x}$ and $W_{\star h}$ contain the weights of the input and recurrent connections, where the subscript $\star$ can either be the input gate i, output gate o, the forget gate f or the memory cell g. On the other hand, $b_\star$ are the biases. The initial values are $s_0 = 0$ and $h_0 = 0$. In addition, $\cdot$ means matrix multiplication, while $*$ means Hadamard product (element-wise product). Note that $g(t)$ is also named as $\tilde{c}(t)$, and $s(t)$ as $c(t)$.

$$g(t) = tanh(W_{gx} \cdot x(t) + W_{gh} \cdot h(t-1) + b_g) \tag{11}$$

$$i(t) = sigmoid(W_{ix} \cdot x(t) + W_{ih} \cdot h(t-1) + b_i) \tag{12}$$

$$f(t) = sigmoid(W_{fx} \cdot x(t) + W_{fh} \cdot h(t-1) + b_f) \tag{13}$$

$$o(t) = sigmoid(W_{ox} \cdot x(t) + W_{oh} \cdot h(t-1) + b_o) \tag{14}$$

$$s(t) = g(t) * i(t) + s(t-1) * f(t) \tag{15}$$

$$h(t) = tanh(s(t)) * o(t) \tag{16}$$

4.9 Summary

This section mainly introduces the FPTT theory which consists of BPTT online formulation and regularization penalty $R(t)$. Intuitively, BPTT online formulation processes a long input sequence in a way that breaks it into multiple smaller sub-sequences and handles each one by BPTT and then updates the parameter. The number of updates equals the number of sub-sequences. $R(t)$ is the extra constraint to stabilize all updates. The paper [2] proves its mathematical foundation. It is shown that FPTT can save computational cost compared with BPTT on sequence modeling tasks, and requires less memory space on both terminal prediction and sequence modeling tasks. Moreover, FPTT is claimed to have a better ability to generalization in terms of accuracy on multiple tasks.

5 FPTT exploration

This section explains explorations and preparations conducted in order to implement FPTT in hardware properly.

5.1 Network Architecture

The default network [2] with which FPTT works includes some advanced techniques such as dropout and gradient clip. These features lead to extreme accuracy and convergence but may not be easy to implement as hardware, so it is necessary to find a relatively small network configuration by which the performance is still reasonable, to simplify the hardware implementation.

First and foremost, in the sFPTT paper [8], it is stated that FPTT only works with RNNs that have gate structure, so the recurrent layer must be LSTM/GRU.

Table 3 lists two configurations. Compared with the default one, the simplified one has fewer layers, a more basic optimizer and no advanced optimization.

Table 3: Configuration: The default vs The simplified

Configuration	Network Structure	Network Size	Optimizer	Dropout	Clips gradient norm
the Default	Input-LSTM-FC-FC	1-128-256-10	Adam	with	with
the Simplified	Input-LSTM-FC	1-128-10	SGD [1]	without	without

[1] With Momentum

The choice of simplified configuration can be validated in table 4. The most accuracy loss is with the dataset CIFAR-10, while on the other two datasets the accuracy loss is negligible. Therefore the simplified network configuration is considered for hardware implementation.

Table 4: Accuracy: The default vs The simplified [1]

Configuration	S-MNIST	PS-MNIST	CIFAR-10
the default	98.67%	94.75%	71.03%
the simplified [2]	98.44%	94.74%	60.04%
accuracy loss	<1%	<1%	≈10%

[1] K used here for both versions is 10.

[2] SGD must be with Momentum, otherwise the accuracy loss can be more than 50%

5.2 Handling of running average in hardware

In the previous section, it is introduced that FPTT consists of two parts, BPTT online formulation with the option K and the regularization penalty $R(t)$, while the latter part can be considered as compensation for the former part. Furthermore, option K offers a trade-off between memory access (area) and memory space (area).

Then it becomes natural to think about whether to implement $R(t)$ as the optional or compulsory data path. In other words, since we may use different K values no matter in design space exploration or real execution, it is important to know if $R(t)$ is necessary for all K values of all datasets involved. So if not, $R(t)$ should be a flexible option to be easily excluded when applicable. To be specific, when $R(t)$ is not critical with some wanted K values, i.e. $R(t)$ can not make more improvement in accuracy based on BPTT online formulation, it should be skipped in computations to save energy and latency.

In [2], it is stated that K is usually set to $\sqrt{T}$ to have reasonable trainability, while the quantitative analysis of trainability versus K with or without $R(t)$ is not given. To investigate this, we first make a hypothesis

as in figure 3 which is the expected relation of achievable precision with the K values with or without $R(t)$. Empirically, for BPTT when the input sequence is longer than 200 steps, gradient vanishing/exploding becomes obvious. This is also the minimal length used for Add Task. Therefore, when K is small the BPTT online formulation with or without $R(t)$ are both expected to perform poorly, as small K leads to relatively long subsequences which may be longer than 200.

Then when K gets a bit larger which makes subsequences much smaller than 200, the guess is that BPTT online formulation alone can achieve reasonable accuracy, as it shows in [2] that the Add task is trained well without $R(t)$.

Nevertheless, when K is large, for example being the same as the sequence length T, the BPTT online formulation is no longer expected to do well because of the intuition that the dependency or feature is split into too many subsequences to be captured as a whole. Moreover, frequent parameter updates can lead to instability that makes it hard to reach the minimum. However, FPTT is expected to make a difference here, as the $R(t)$ in the loss function is designed to provide stability during training.

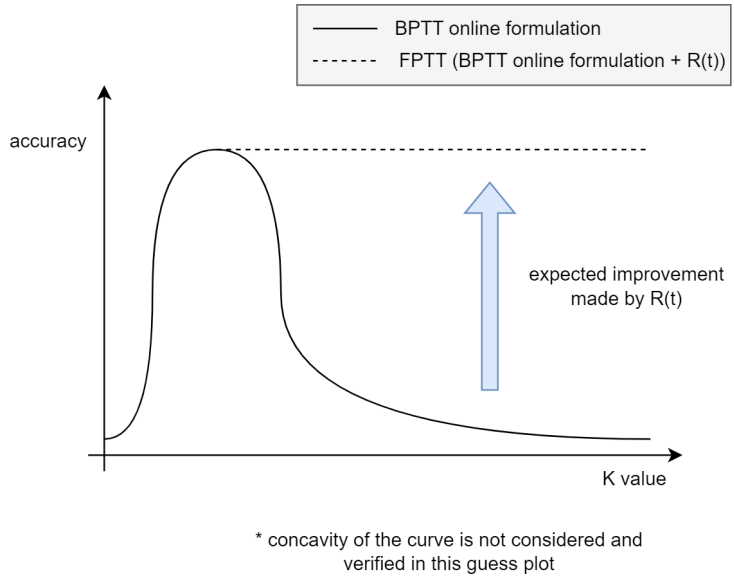

Figure 3: Hypothesis plot of K vs accuracy

We try to validate the hypothesis aforementioned by the S-MNIST task with eight different K values, while the result is against the hypothesis mainly when K is large. It is clear that when K is 392 or 784, which means the length of each subsequence is 2 or 1, $R(t)$ cannot bring improvement in accuracy. As this report aims at digital implementation, further theoretical research to explain this issue is out of scope. If time permits, we may perform validations on sequence modeling tasks like language modeling.

Although the result of the experiments is slightly out of expectation, we still have the conclusion that $R(t)$ should be implemented as the optional data path in the hardware routine.

Table 5: Actual accuracy[1] achieved at epoch 100 by eight K values on S-MNIST task (784 steps)

Loss Function	K=1	K=2	K=4	K=7	K=14	K=98	K=392	K=784
clf[2]	11.35%	42.90%	39.70%	96.62%	98.28%	96.39%	90.89%	83.91%
clf[2]+ oracle	11.35%	97.46%	90.77%	97.69%	98.47%	96.63%	90.65%	80.14%
clf[2]+ oracle + R[3]	11.35%	84.93%	95.39%	97.52%	98.2%	95.99%	91.92%	83.40%

[1] learning rate is 0.01, and it decays at epoch [50, 75, 90] by 0.1
[2] classification loss
[3] regularization penalty $R(t)$

5.3 Manual Derivation of backward pass

The code with FPTT paper [2] uses PyTorch so the gradient calculation is done automatically on GPUs. However, for edge computing, running PyTorch with GPUs is costly. Therefore, it is necessary to find the deterministic and synthesizable data path for gradient calculations for hardware implementation.

FPTT theory defines the loss in two parts, namely the common loss and the regularization penalty $R(t)$. The common loss is the difference between the prediction and the label, which is a complex function of the network

parameters, depending on the network architecture. Especially for terminal prediction problems, FPTT theory introduces the oracle loss. Then the common loss is the classification loss and oracle loss, while the label is the combination of expected output and $Q(\overline{y})$ as it is in definition 10. The derivative of common loss with regard to parameters, namely the gradient, is found by BPTT, as it is too complex to be solved by mathematical analysis.

The regularization penalty $R_{(t)}$ is the difference between the parameters and the running average of the parameters, which is relatively more straightforward than the common loss. Its derivation can be found by direct mathematical analysis of it without BPTT, and this part is introduced in the next section along with the FPTT routine.

The derivation of common loss is dependent on the network architecture. Here, the simplified network version mentioned before is used to show how to find gradients of the common loss by BPTT. To be specific, the network has a three-layer structure of size 1x128x10, assuming the target task is S-MNIST (grayscale).

Layer 0 streams the input sequence so that one element of a pixel enters the network at every time step. For RGB tasks like CIFAR-10 in which each element has 3 pixels, the input dimension would be 3. Layer 1 is a layer of 128 LSTM cells, and the structure of the LSTM is defined in the previous section from equation 11 to 16. Layer 2 consists of 10 cells performing the linear function and then the non-linear softmax.

With the concept of each layer being clear, the forward pass of the simplified network can be defined from equation 19 to 26. Note that concatenation is performed on input and hidden states, as well as input and hidden weight matrix separately, for clearer expression as is in equation 17 and 18.

$$xc(t) = [x(t), h(t-1)] \tag{17}$$

$$W_{l1\star} = [W_{l1\star x}, W_{l1\star h}](\star = i, f, g, o) \tag{18}$$

$$g_{l1}(t) = tanh(ginput_{l1}(t)); \qquad ginput_{l1}(t) = \mathbf{W}_{l1g} \cdot xc(t) + b_{l1g} \tag{19}$$

$$i_{l1}(t) = sigmoid(iinput_{l1}(t)); \qquad iinput_{l1}(t) = \mathbf{W}_{l1i} \cdot xc(t) + b_{l1i} \tag{20}$$

$$f_{l1}(t) = sigmoid(finput_{l1}(t)); \qquad finput_{l1}(t) = \mathbf{W}_{l1f} \cdot xc(t) + b_{l1f} \tag{21}$$

$$o_{l1}(t) = sigmoid(oinput_{l1}(t)); \qquad oinput_{l1}(t) = \mathbf{W}_{l1o} \cdot xc(t) + b_{l1o} \tag{22}$$

$$s_{l1}(t) = g_{l1}(t) * i_{l1}(t) + s_{l1}(t-1) * f_{l1}(t) \tag{23}$$

$$h_{l1}(t) = tanh(s_{l1}(t)) * o_{l1}(t) \tag{24}$$

$$h_{l2}(t) = \mathbf{W}_{l2} \cdot h_{l1}(t) + b_{l2} \tag{25}$$

$$\hat{y} = softmax(h_{l2}(t)) \tag{26}$$

Then, the derivative of common loss w.r.t. parameters can be found by BPTT, including the weight and the bias.

$$\frac{\partial L(t)}{\partial W_\star}; \qquad \frac{\partial L(t)}{\partial b_\star} \qquad (\star = l1i, l1o, l1f, l1g, l2) \tag{27}$$

Assuming that the loss function is the cross-entropy loss and the label is y. First of all, we need to find the partial derivative of cross-entropy loss with respect to the input of the softmax function. This is a nontrivial problem but is well explained in post[2] in details and the result is in equation 28. Note that in the following equations, the potential transposition of matrices is not mentioned for simplicity in expression, but the details can be found in the code implementation[3].

$$\frac{\partial L(t)}{\partial h_{l2}(t)} = \hat{y} - y \tag{28}$$

Immediately we can have the derivative of loss w.r.t. the weight and the bias in layer 2. Layer 2 is non-recurrent, so the layer output only depends on the layer input at the same step. In such a way, no BPTT is needed to find the influence of weight/bias from previous time steps. With equation 28 ready, we need one

[2]https://www.mldawn.com/back-propagation-with-cross-entropy-and-softmax/

[3]https://gitlab.tue.nl/20220018/sfptt_c/-/tree/main/20230703

more step to reach equation 29 and 30, by calculating the gradient of the hidden state in layer to w.r.t. the parameter in layer 2. Note that dimensions need to be compressed to 1 for batch training in equation 30.

$$\frac{\partial L(t)}{\partial \mathbf{W}_{l2}} = \frac{\partial L(t)}{\partial h_{l2}(t)} \cdot \frac{\partial h_{l2}(t)}{\partial W_{l2}} = (\hat{y} - y) \cdot h_{l1}(t) \tag{29}$$

$$\frac{\partial L(t)}{\partial b_{l2}} = \frac{\partial L(t)}{\partial h_{l2}(t)} \cdot \frac{\partial h_{l2}(t)}{\partial b_{l2}} = \frac{\partial L(t)}{\partial h_{l2}(t)} = \hat{y} - y \tag{30}$$

Meanwhile, the derivative of loss w.r.t. $h_{l1}(t)$ is also obtained.

$$\frac{\partial L(t)}{\partial h_{l1}(t)} = \frac{\partial L(t)}{\partial h_{l2}(t)} \cdot \frac{\partial h_{l2}(t)}{\partial h_{l1}(t)} = (\hat{y} - y) \cdot W_{l2} \tag{31}$$

As an LSTM layer, the output of Layer 1 at time step t depends on all hidden states prior to t. In addition, hidden states at every time step depend on weight and bias. As it shows in equation 32, to find the derivative of loss $L(t)$ with respect to the parameters in layer 1, by BPTT we need to go through all steps earlier than t to accumulate the effect, with t being the starting point, i.e., the last step. In terms of the earliest step (ending point) to refer to, FPTT(-K) ends up with the first step in the sub-sequence. So, if each sub-sequence has one step only, there would be one term in equation 32. However, traditional BPTT training does not finish the backtracking until the very first step of the entire input sequence.

$$\frac{\partial L(t)}{\partial W_{l1}} = \sum_{i=latest}^{earliest} \frac{\partial L(t)}{\partial h_{l1}(i)} \cdot \frac{\partial h_{l1}(i)}{\partial W_{l1}} \tag{32}$$

Expanding the summation, we have equation 33.

$$\frac{\partial L(t)}{\partial W_{l1}} = \frac{\partial L(t)}{\partial h_{l1}(t)} \cdot \frac{\partial h_{l1}(t)}{\partial W_{l1}} + \frac{\partial L(t)}{\partial h_{l1}(t-1)} \cdot \frac{\partial h_{l1}(t-1)}{\partial W_{l1}} + \frac{\partial L(t)}{\partial h_{l1}(t-2)} \cdot \frac{\partial h_{l1}(t-2)}{\partial W_{l1}} + ... \tag{33}$$

Accessing an arbitrary $h_{l1}(i)$ by BPTT is through all hidden states generated in later time steps, i.e. $h_{l1}(t), h_{l1}(t-1), ..., h_{l1}(i+1)$, as $h_{l1}(i)$ influence them in the forward pass.

$$\frac{\partial L(t)}{\partial W_{l1}} = \frac{\partial L(t)}{\partial h_{l1}(t)} \cdot \frac{\partial h_{l1}(t)}{\partial W_{l1}} + \frac{\partial L(t)}{\partial h_{l1}(t)} \cdot \frac{\partial h_{l1}(t)}{\partial h_{l1}(t-1)} \cdot \frac{\partial h_{l1}(t-1)}{\partial W_{l1}} + \frac{\partial L(t)}{\partial h_{l1}(t)} \cdot \frac{\partial h_{l1}(t)}{\partial h_{l1}(t-1)} \cdot \frac{\partial h_{l1}(t-1)}{\partial h_{l1}(t-2)} \cdot \frac{\partial h_{l1}(t-2)}{\partial W_{l1}} + ... \tag{34}$$

Furthermore, the calculation of $\frac{\partial h_{l1}(t)}{\partial W_{l1}}$ in equation 34 follows a fixed routine for any t ranging from 0 to $T-1$, as well as $\frac{\partial h_{l1}(t)}{\partial h_{l1}(t-1)} (1 \leq t < T)$. So, to evaluate equation 34 is to find $\frac{\partial h_{l1}(t)}{\partial W_{l1}}$ at a specific t, and then find $\frac{\partial h_{l1}(t)}{\partial h_{l1}(t-1)} (1 \leq t < T)$ to be able to iterate the fixed routine and accumulate, given $\frac{\partial L(t)}{\partial h_{l1}(t)}$.

We focus on $\frac{\partial L(t)}{\partial h_{l1}(t)} \cdot \frac{\partial h_{l1}(t)}{\partial W_{l1}}$. The unfolded forward pass of LSTM in terms of time is shown in the figure 4. It can be observed that $i_{l1}(t)$, $f_{l1}(t)$ and $g_{l1}(t)$ contribute to $h_{l1}(t)$ in the same way via $s_{l1}(t)$, while $o_{l1}(t)$ goes to $h_{l1}(t)$ directly. Thus, $\frac{\partial L(t)}{\partial W_{l1o}}$ and $\frac{\partial L(t)}{\partial b_{l1o}}$ can be solved as follows from equation 35 to 38.

$$\frac{\partial L(t)}{\partial o_{l1}(t)} = \frac{\partial L(t)}{\partial h_{l1}(t)} * \frac{\partial h_{l1}(t)}{\partial o_{l1}(t)} = \frac{\partial L(t)}{\partial h_{l1}(t)} * tanh(s_{l1}(t)) \tag{35}$$

$$\frac{\partial L(t)}{\partial oinput_{l1}(t)} = \frac{\partial L(t)}{\partial o_{l1}(t)} * \frac{\partial o_{l1}(t)}{\partial oinput_{l1}(t)} = \frac{\partial L(t)}{\partial o_{l1}(t)} * (o_{l1}(t) * (1 - o_{l1}(t))) \tag{36}$$

$$\frac{\partial L(t)}{\partial W_{l1o}} = \frac{\partial L(t)}{\partial oinput_{l1}(t)} \cdot \frac{\partial oinput_{l1}(t)}{\partial W_{l1o}} = \frac{\partial L(t)}{\partial oinput_{l1}(t)} \cdot xc(t) \tag{37}$$

$$\frac{\partial L(t)}{\partial b_{l1o}} = \frac{\partial L(t)}{\partial oinput_{l1}(t)} \cdot \frac{\partial oinput_{l1}(t)}{\partial b_{l1o}} = \frac{\partial L(t)}{\partial oinput_{l1}(t)} \tag{38}$$

To calculate $\frac{\partial h_{l1}(t)}{\partial W_{l1\star}}$ and $\frac{\partial h_{l1}(t)}{\partial b_{l1\star}} (\star = i, f, g)$, $s_l 1(t)$ needs to be considered and it follows equation 39.

$$\frac{\partial h_{l1}(t)}{\partial X_{l1\star}} = \sum_{i=latest}^{earliest} \frac{\partial h_{l1}(t)}{\partial s_{l1}(i)} \cdot \frac{\partial s_{l1}(i)}{\partial X_{l1\star}} \qquad (X = W, b; \star = i, f, g) \tag{39}$$

Expanding equation 39, we obtain 40.

$$\frac{\partial h_{l1}(t)}{\partial W_{l1\star}} = \frac{\partial h_{l1}(t)}{\partial s_{l1}(t)} \cdot \frac{\partial s_{l1}(t)}{\partial W_{l1\star}} + \frac{\partial h_{l1}(t)}{\partial s_{l1}(t-1)} \cdot \frac{\partial s_{l1}(t-1)}{\partial W_{l1\star}} + \frac{\partial h_{l1}(t)}{\partial s_{l1}(t-2)} \cdot \frac{\partial s_{l1}(t-2)}{\partial W_{l1\star}} + ... \qquad (\star = i, f, g) \tag{40}$$

Calculation of $\frac{\partial h_{l1}(t)}{\partial s_{l1}(i)}(i<t)$ needs special attention, as $s_{l1}(i)$ can affect the following states in two directions, as shown in equation 41. Here we only consider the first direction, i.e., the direct path between $s_{l1}(t-1)$, $s_{l1}(t)$ and $s_{l1}(t+1)$ ... (figure 4), because $\frac{\partial h_{l1}(t+1)}{\partial h_{l1}(t)}$ in the second direction is already covered in other terms of equation 34.

$$\frac{\partial h_{l1}(t+1)}{\partial s_{l1}(t)} = \frac{\partial h_{l1}(t+1)}{\partial s_{l1}(t+1)} \cdot \frac{\partial s_{l1}(t+1)}{\partial s_{l1}(t)} + \frac{\partial h_{l1}(t+1)}{\partial h_{l1}(t)} \cdot \frac{\partial h_{l1}(t)}{\partial s_{l1}(t)} \tag{41}$$

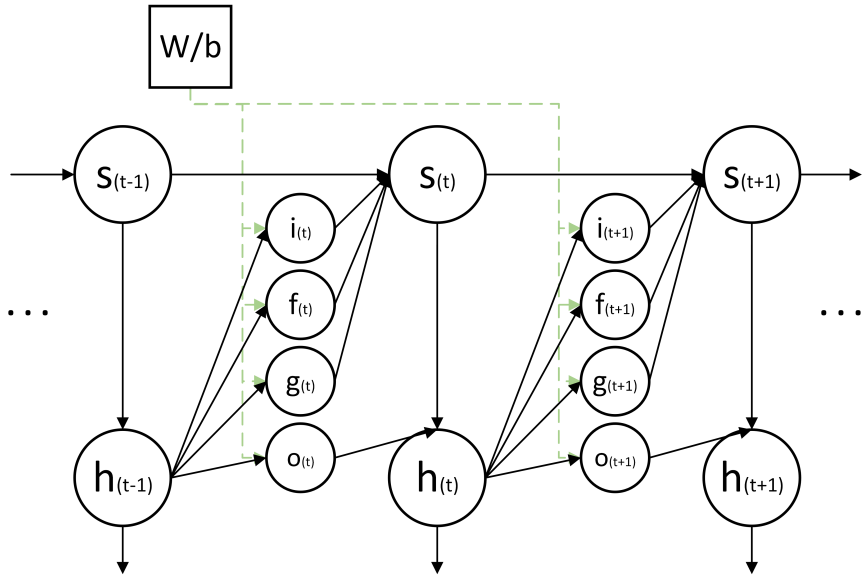

Figure 4: LSTM unrolling

Then equation 40 can be transformed into 42.

$$\frac{\partial h_{l1}(t)}{\partial W_{l1\star}} = \frac{\partial h_{l1}(t)}{\partial s_{l1}(t)} \cdot \frac{\partial s_{l1}(t)}{\partial W_{l1\star}} + \frac{\partial h_{l1}(t)}{\partial s_{l1}(t)} \cdot \frac{\partial s_{l1}(t)}{\partial s_{l1}(t-1)} \cdot \frac{\partial s_{l1}(t-1)}{\partial W_{l1\star}} + ... \qquad (\star = i, f, g) \tag{42}$$

Similar to evaluation of equation 34, we evaluate quation 42 by means of the fixed routine which solves $\frac{\partial h_{l1}(t)}{\partial s_{l1}(t)} \cdot \frac{\partial s_{l1}(t)}{\partial W_{l1\star}}(\star = i, f, g)$ and $\frac{\partial s_{l1}(t)}{\partial s_{l1}(t-1)}$ while iterate routine and accumulate, given $\frac{\partial L(t)}{\partial h_{l1}(t)}$. But here we solve $\frac{\partial s_{l1}(t)}{\partial s_{l1}(t-1)}$ first, since it is more straightforward as shown in equation 43.

$$\frac{\partial s_{l1}(t)}{\partial s_{l1}(t-1)} = f_{l1}(t) \tag{43}$$

Calculation of $\frac{\partial L(t)}{\partial X_{l1\star}}(X = W, b; \star = i, f, g)$ is via $s_{l1}(t)$, so we need to have $\frac{\partial L(t)}{\partial s_{l1}(t)}$ in equation 44.

$$\frac{\partial L(t)}{\partial s_{l1}(t)} = \frac{\partial L(t)}{\partial h_{l1}(t)} * \frac{\partial h_{l1}(t)}{\partial tanh(s_{l1}(t))} * \frac{\partial tanh(s_{l1}(t))}{\partial s_{l1}(t)} = \frac{\partial L(t)}{\partial h_{l1}(t)} * o_{l1}(t) * (1 - tanh^2(s_{l1}(t))) \tag{44}$$

Now we can solve $\frac{\partial L(t)}{\partial \star_{l1}(t)}$ $(\star = i, f, g)$.

$$\frac{\partial L(t)}{\partial i_{l1}(t)} = \frac{\partial L(t)}{\partial s_{l1}(t)} * \frac{\partial s_{l1}(t)}{\partial i_{l1}(t)} = \frac{\partial L(t)}{\partial s_{l1}(t)} * g_{l1}(t) \tag{45}$$

$$\frac{\partial L(t)}{\partial g_{l1}(t)} = \frac{\partial L(t)}{\partial s_{l1}(t)} * \frac{\partial s_{l1}(t)}{\partial g_{l1}(t)} = \frac{\partial L(t)}{\partial s_{l1}(t)} * i_{l1}(t) \tag{46}$$

$$\frac{\partial L(t)}{\partial f_{l1}(t)} = \frac{\partial L(t)}{\partial s_{l1}(t)} * \frac{\partial s_{l1}(t)}{\partial f_{l1}(t)} = \frac{\partial L(t)}{\partial s_{l1}(t)} * s_{l1}(t-1) \tag{47}$$

In the same way, we obtain $\frac{\partial L(t)}{\partial W_{l1o}}$ and $\frac{\partial L(t)}{\partial b_{l1o}}$, so we can have $\frac{\partial L(t)}{\partial W_{l1\star}}$ and $\frac{\partial L(t)}{\partial b_{l1\star}}$ $(\star = i, f, g)$.

$$\frac{\partial L(t)}{\partial iinput_{l1}(t)} = \frac{\partial L(t)}{\partial i_{l1}(t)} * \frac{\partial i_{l1}(t)}{\partial iinput_{l1}(t)} = \frac{\partial L(t)}{\partial i_{l1}(t)} * (i_{l1}(t) * (1 - i_{l1}(t))) \tag{48}$$

$$\frac{\partial L(t)}{\partial finput_{l1}(t)} = \frac{\partial L(t)}{\partial f_{l1}(t)} * \frac{\partial f_{l1}(t)}{\partial finput_{l1}(t)} = \frac{\partial L(t)}{\partial f_{l1}(t)} * (f_{l1}(t) * (1 - f_{l1}(t))) \tag{49}$$

$$\frac{\partial L(t)}{\partial ginput_{l1}(t)} = \frac{\partial L(t)}{\partial g_{l1}(t)} * \frac{\partial g_{l1}(t)}{\partial ginput_{l1}(t)} = \frac{\partial L(t)}{\partial g_{l1}(t)} * (1 - g_{l1}^2(t)) \tag{50}$$

$$\frac{\partial L(t)}{\partial W_{l1\star}} = \frac{\partial L(t)}{\partial \star input_{l1}(t)} \cdot \frac{\partial \star input_{l1}(t)}{\partial W_{l1\star}} = \frac{\partial L(t)}{\partial \star input_{l1}(t)} \cdot xc(t) \qquad (\star = i, f, g) \tag{51}$$

$$\frac{\partial L(t)}{\partial b_{l1\star}} = \frac{\partial L(t)}{\partial \star input_{l1}(t)} \cdot \frac{\partial \star input_{l1}(t)}{\partial b_{l1\star}} = \frac{\partial L(t)}{\partial \star input_{l1}(t)} \qquad (\star = i, f, g) \tag{52}$$

With equation 51, 52 and 43 we can evaluate equation 42 by iteration and accumulation. Furthermore, together with equation 37 and 38 they conclude the fixed routine of derivative of loss w.r.t. all parameters in layer 1 at a specific $h_{l1}(t)$, namely $\frac{\partial h_{l1}(t)}{\partial X_{l1}}(X = W, b)$, given $\frac{\partial L(t)}{\partial h_{l1}(t)}$. Nevertheless, we also need to solve $\frac{\partial h_{l1}(t)}{\partial h_{l1}(t-1)}$ to iterate that routine as mentioned in equation 34.

$h_{l1}(t)$ is influenced by $h_{l1}(t-1)$ two ways. For one thing, $h_{l1}(t-1)$ contributes to $s_{l1}(t)$ by $i_{l1}(t)$, $f_{l1}(t)$ and $g_{l1}(t)$. For another, $h_{l1}(t-1)$ effects $o_{l1}(t)$. Therefore, we have equation 53.

$$\frac{\partial L(t)}{\partial h_{l1}(t-1)} = \frac{\partial L(t)}{\partial h_{l1}(t)} \cdot \frac{\partial h_{l1}(t)}{\partial h_{l1}(t-1)} = \frac{\partial L(t)}{\partial h_{l1}(t)} \cdot (\frac{\partial h_{l1}(t)}{\partial iinput_{l1}(t)} \cdot \frac{\partial iinput_{l1}(t)}{\partial h_{l1}(t-1)} +$$

$$\frac{\partial h_{l1}(t)}{\partial finput_{l1}(t)} \cdot \frac{\partial finput_{l1}(t)}{\partial h_{l1}(t-1)} + \frac{\partial h_{l1}(t)}{\partial ginput_{l1}(t)} \cdot \frac{\partial ginput_{l1}(t)}{\partial h_{l1}(t-1)} + \frac{\partial h_{l1}(t)}{\partial oinput_{l1}(t)} \cdot \frac{\partial oinput_{l1}(t)}{\partial h_{l1}(t-1)}) \tag{53}$$

Get the $\frac{\partial L(t)}{\partial h_{l1}(t)}$ into the bracket, we have equation 54.

$$\frac{\partial L(t)}{\partial h_{l1}(t-1)} = \frac{\partial L(t)}{\partial h_{l1}(t)} \cdot \frac{\partial h_{l1}(t)}{\partial h_{l1}(t-1)} = \frac{\partial L(t)}{\partial iinput_{l1}(t)} \cdot \frac{\partial iinput_{l1}(t)}{\partial h_{l1}(t-1)} +$$

$$\frac{\partial L(t)}{\partial finput_{l1}(t)} \cdot \frac{\partial finput_{l1}(t)}{\partial h_{l1}(t-1)} + \frac{\partial L(t)}{\partial ginput_{l1}(t)} \cdot \frac{\partial ginput_{l1}(t)}{\partial h_{l1}(t-1)} + \frac{\partial L(t)}{\partial oinput_{l1}(t)} \cdot \frac{\partial oinput_{l1}(t)}{\partial h_{l1}(t-1)} \tag{54}$$

As $\frac{\partial L(t)}{\partial \star input_{l1}(t)}(\star = i, f, g, o)$ are derived in equation 48, 49, 50 and 36, equation 54 can be evaluated.

$$\frac{\partial L(t)}{\partial h_{l1}(t-1)} = \frac{\partial L(t)}{\partial iinput_{l1}(t)} \cdot W_{l1ih} + \frac{\partial L(t)}{\partial finput_{l1}(t)} \cdot W_{l1fh} + \frac{\partial L(t)}{\partial ginput_{l1}(t)} \cdot W_{l1gh} + \frac{\partial L(t)}{\partial oinput_{l1}(t)} \cdot W_{l1oh} \tag{55}$$

In this way, we can have $\frac{\partial L(t)}{\partial h_{l1}(i)}$ for any $i < t$.Combining with how we obtain $\frac{\partial h_{l1}(i)}{W_{l1}}$ and accumulation, we evaluate equation 32.

5.4 FPTT routine

In this subsection, we find the derivative of the regularization penalty $R(t)$ w.r.t. parameters. Along with the derivative of common loss w.r.t. parameters, we complete the gradient calculation. Then, we define the routine to update the parameters by calculated gradients, update the running average and the lambda.

The derivation of the gradient of $R(t)$ is presented in paper [2], so we follow it here. Considering the situation where the number of gradient steps of the loss approaches infinity, minimizing loss function is described as 56, in which terms are explained in the previous section 4.4.

$$W_{t+1} = arg_W min(l_t(W) + \frac{\alpha}{2}\|W - \overline{W}_t - \frac{1}{2\alpha}\nabla_{l_{t-1}}(W_t)\|^2) \tag{56}$$

Taking gradient w.r.t. W results in the following dynamics in equation 57. W_{t+1} is the value of the weight that results in the minimum of the loss function at time step t.

$$\nabla_{l_t}(W_{t+1}) - \nabla_{l_{t-1}}(W_t) + \alpha(W_{t+1} - \overline{W}_t) = 0 \tag{57}$$

Generally for neural networks, it is hard to find W_{t+1} directly which can minimize the loss. Instead, gradient descent should be used to approach the minimal loss function, and the gradient of loss w.r.t. to the weight and optionally the bias at time step t can be defined if W_{t+1} is replaced by W_t at the left hand side of equation 57. Note that $\nabla_{l_t}(W_t)$ is the gradient of common loss obtained by BPTT which is explained in the last subsection, while $-\nabla_{l_{t-1}}(W_t) + \alpha(W_t - \overline{W}_t)$ is the gradient of regularization penalty $R(t)$, which is calculated in-place.

$$\frac{\partial L(t)}{\partial W} = \nabla_{l_t}(W_t) - \nabla_{l_{t-1}}(W_t) + \alpha(W_t - \overline{W}_t) \tag{58}$$

Moreover, 59 is the update rule for $\overline{W}$ which is mentioned before.

$$\overline{W}_{t+1} = \frac{1}{2}(\overline{W}_t + W_{t+1}) - \frac{1}{2\alpha}\nabla_{l_t}(W_{t+1}) \tag{59}$$

With 59 and 58, we know how to update the weights and the running mean of the weights, respectively, while the remaining problem is to calculate $\nabla_{l_t}(W_{t+1})$ in 59 and $\nabla_{l_{t-1}}(W_t)$ in 58. It can be observed that these two terms are basically the same but with different subscripts. In paper [2], such a term is obtained by a running estimate $\lambda_t(\lambda_0 = 0)$

$$\lambda_{t+1} = \lambda_t - \alpha(W_{t+1} - \overline{W_t}) \tag{60}$$

With equations 58, 60 and 59 defined, we can design the routine in the program. For each time step (see FPTT algorithm list), once the network performs the forward pass for one time step, the following steps need to be done before proceeding to the next forward pass.

1. Evaluate 58 to get the gradient of entire loss function w.r.t. parameters;
2. Multiply the gradient with the learning rate, and subtract the result from current weights W_t. Then we have the new weights W_{t+1};
3. according to 60, update the λ by new weights W_{t+1} and current running mean $\overline{W_t}$;
4. according to 59, update the $\overline{W}$ by new λ_{t+1}, new weights W_{t+1} and current running mean $\overline{W_t}$.

In Python, the FPTT routine can be implemented as in 1, assuming the gradient of common loss $\nabla_{l_t}(W_t)$ is ready.

```
# variable naming

# param:    the parameter of the network, including the weight and optionally the bias
# grad:     the gradient of common loss, obtained by BPTT
# lbd:      lambda
# rmean:    running average/mean of the parameter

# alpha:    coefficient in FPTT theory, hyper-parameter of the network, normally constant in the training
# ita:      the inverse of two times alpha for short, 1/(2*alpha)
# lr:       learning rate, hyper-parameter of the network, can decay in the training


param -= lr * (grad - lbd + alpha * (param - rmean))
lbd -= lbd - alpha * (param - rmean)
rmean = 0.5*(param + rmean) - ita * lbd
```

Listing 1: FPTT calculation core part

In 1 it can be observed that operations are element-wise and independent from the network architecture, as they affect each element in the matrix separately. To be specific, all multiplications are element-wise, i.e. no matrix multiplications and the addition and subtractions are also element-wise. Therefore, all elements in the matrix $\nabla_{l_t}(W_t)$, W, $\overline{W}$ and λ can be executed in the same routine in parallel.

5.5 Summary

In this section, we first find a relatively small network configuration for hardware implementation. Then, by hypothesis and experiment we investigate the effect of $R(t)$. We find that $R(t)$ is not always necessary, so it is reasonable to implement that as an optional rather than mandatory data path in hardware. Also, as K in FPTT-K offers a trade-off between memory access (energy) and memory space (area), it is interesting to implement the scalable K in hardware so that design space exploration can be performed to find the optimal PPA.

Afterward, we solve the calculation for the gradient of common loss and $R(t)$ w.r.t. the parameter separately. Last we find the routine that coordinates the regular backward pass and FPTT logic. The outcome of this stage is a training script in Python, its correctness can be shown in table 6. It can be observed that the script is functioning but needs to be improved further.

To conclude, all computations are determined so that the routine can be implemented as hardware and the hardware architecture can be proposed.

Table 6: The result of self-design FPTT training script on MNIST task

The way of processing	self-designed version	Pytorch version
streaming by rows	94%	98%
streaming by pixels	78%	88%

6 Hardware Architecture Plan

Based on previous work, a tentative hardware system is proposed and analyzed quantitatively.

6.1 system and analysis

The dataflow of the system is in figure 5. The FPTT training system is based on the traditional BPTT training system (in dashed box), along with the dedicated function and memory for FPTT calculation. To start with, the parameter(weights & biases) and input sequence are used for the network forward pass. The generated network states are stored in the memory, as they will be used for the backward pass. The backward pass takes the parameter, network states and the labels (may include oracle distribution) to calculate the gradient. Then, the FPTT logic receives four types of input, i.e., the gradient, the lambda, the running mean and the parameter then updates the lambda, the running mean and the parameter.

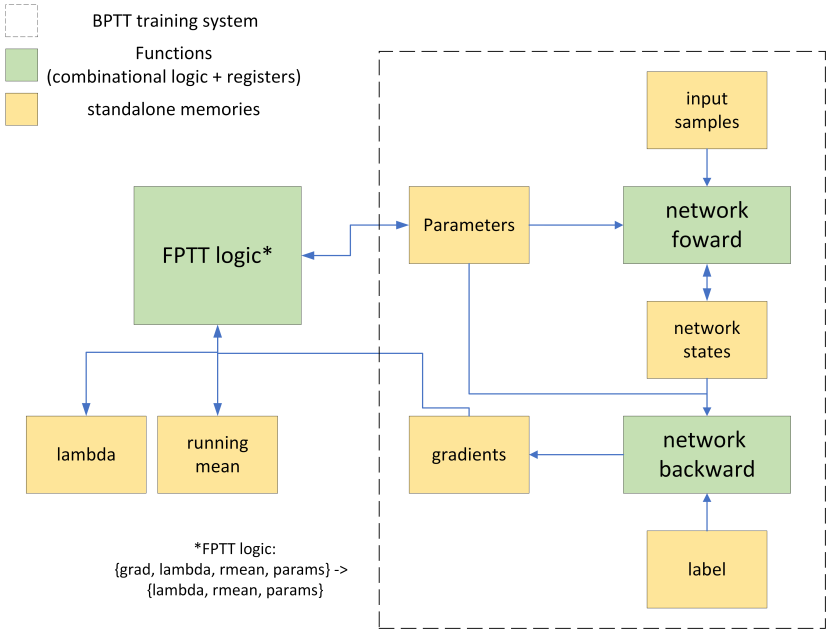

Figure 5: Dataflow of the system

The application/dataflow is implemented with Python code using NumPy (i.e. no parallel acceleration), which is the starting point and baseline of the hardware implementation. So, we analyze this application in terms of memory requirement, operation counts and execution time of each function. In such a way, we can be quantitatively familiar with the system specification, find the critical part and then decide the hardware solution. Here we still take the simplified network configuration as an example.

- 3-layer network of the input layer, LSTM layer and fully connected output layer (1x128x10)
- The application is fully flattened MNIST, therefore the sequence of 784 time steps
- FPTT-28 (784=28x28, so 28 sub-sequences, each has 28 time steps)
- Batch size is 100
- the data size of each element is float32 (4 Bytes)

The estimation of the memory requirement of the system is listed in table 7. `Parameters` consist of the weight and the bias in the system. In addition, `Lambda`, `Running Mean` and `Gradients` all have a one-to-one correspondence with `Parameters`, so these four memories are of the same size. Note that `Gradients` are

calculated by network backward pass in a sequential way, thus it may be further optimized later by means of streaming processing. The size of the remaining five memories or register groups is dependent on the batch size in the training process. Among those, the size of the memory for `Network States` directly benefits from the FPTT algorithm. In detail, the system only needs to store 28 time steps of states to be used in the backward pass for a specific sub-sequence. When this sub-sequence is processed, the memory can be released and then it can store the states of the next sub-sequence. By contrast, the BPTT algorithm requires saving all time steps of states created by the entire input sequence. In this case, `Network States` alone can take 240 MB (28*8600/1024). Therefore, saved memory by FPTT enables larger batch sizes no matter for training on GPUs or on accelerators.

Table 7: Estimated memory requirement of the system

Name		Size	Note
Memory	Parameters	265 KB	((128*1+1+128*128+128)*4+(10*128+10))*4B
	Lambda	265 KB	identical to the above
	Running Mean	265 KB	identical to the above
	Gradients	265 KB	identical to the above
	Input Sample [1]	400 B	28*100*4 B
	Label[1]	440 B	(100 + 10)*4B
	Network States[1]	8600 KB	28*(128*6 + 10*2)*100 *4B
Register[2]	Network Forward[1]	200 KB	128*4*100*4B
	Network Backward[1]	500 KB	(128*10 + 10*1)*100*4B
Total		≈ **10.2MB**	≈ 1.2 MB/3 MB if batch size is 1/10

[1] depends on the batch size
[2] tentative values here, will depend on pipeline design

Then, the estimation of operation count in one Epoch performed by each function block is listed in table 8. Note that the full MNIST dataset has 60K samples, so there needs to be 600 iterations to achieve one Epoch using a batch size of 100. It can be observed that the calculation of FPTT logic is just a small fraction of the workload, and the network forward pass and backward pass are the critical part that dominates the energy consumption.

Table 8: Estimated operation counts required by each function block

Function Block	MUL/DIV	ADD/SUB	non-linear activation
FPTT logic	5.7 G	9.1 G	N/A
Network forward	3185.2 G	3167.4 G	3 G
Network backward	5609.4 G	5470.9 G	12 G

Furthermore, the execution time of each function is listed in table 9, for sub-sequence of 28 time steps when the batch size is 100. It is clear that the execution time of FPTT logic also takes a very small piece of the total time consumption, i.e., 0.11%.

Table 9: Execution time of each function block of Python program

Function Block	Execution Time (sec)	percentage
FPTT logic	0.000367	0.11%
Network forward	0.138306	39.73%
Network backward	0.209401	60.16%

The amount of workload decides the power consumption, while the execution time reflects the processing latency (usually can be accelerated when the area is paid) if the application is to be implemented as hardware. Thus to realize the full system and make it work efficiently in terms of power and latency, the network forward and backward is critical and worth implementation in a fixed data path, in spite of the complexity in logic design which is obvious by the BPTT derivation explained before.

On the other hand, although the FPTT logic seems trivial with workload and execution time, it is also interesting to realize it as a fixed data path due to its general applicability. Since FPTT logic is only related to the parameters, and is independent of network architecture (forward pass), it can be a flexible plug-in for any type of RNN architecture. Also, it is the greatest innovation in the FPTT algorithm. To sum up, it looks

natural to design the full system in dedicated hardware, though the biggest problem, i.e. the complexity of backward pass in logic design is unavoidable.

Another point of view is to design specialized hardware for FPTT logic only, while network forward and backward passes are executed on some relatively general-purpose processor like CPU. In such a way, we can get rid of the difficulties in the logic design network forward/backward pass by directly running high-level programs on it. Furthermore, it is normally easier to develop RNNs in high-level languages like C than HDL, then it becomes possible to showcase the ability of generalization of FPTT with multiple RNNs. We decided to take this option, encouraged by flexibility and the lower complexity in design.

6.2 FPTT logic design and varities

In this subsection logic design of FPTT calculation is proposed. Previously, it is explained that all operations in the FPTT logic are element-wise, i.e. no dependency in between. In such a way, every parameter can be processed in parallel ideally if unlimited resources are provided. However in practice, it is suitable to design a vector processor to handle a batch of parameters in parallel each time, and by multiple iterations, all parameters can be processed.

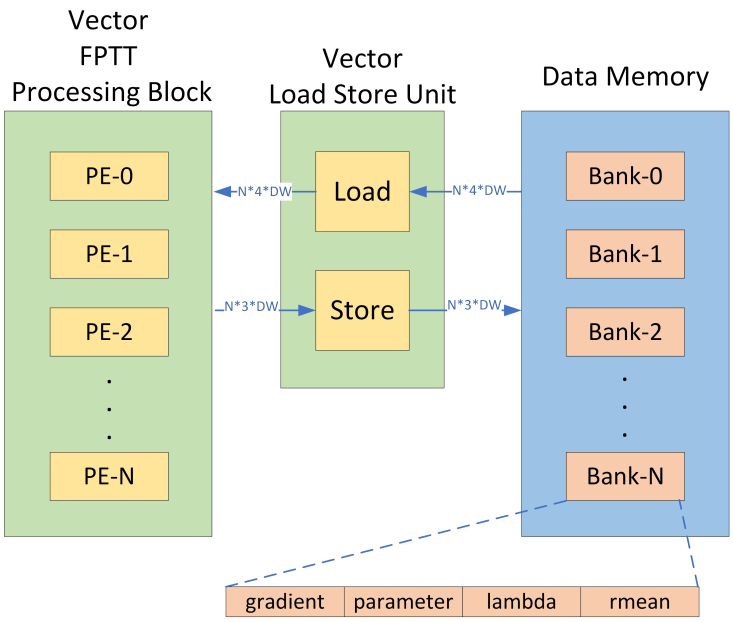

Figure 6: Tentative FPTT subsystem

Therefore, a naive FPTT subsystem is proposed in figure 6. All PEs in the vector FPTT processing block are identical, each of them receives one group of {gradient, parameter, lambda, running mean} from a specific line in the data memory through the load module. Then they process the data and return the updated {parameter, lambda, running mean} to the same line memory via the store module. The number of PEs and the bank of memory banks should match, both being N, to exploit the parallelism. Also, the Load Module has a bandwidth of $N * 4 * DataWidth$ while the Store Module can have less bandwidth of $N * 3 * DataWidth$ if possible. Note that the number of processing elements (PE) in the vector is changeable. As it offers a trade-off between latency and area, it will be determined after the design space exploration.

Afterward, we investigate the logic design of a single PE. Basically, we convert the following three statements of high-level language (list 2) to hardware implementation. To that end, timing should be considered carefully to avoid long combinational paths which can limit work frequency. This is usually achieved by breaking the long series of operations and storing the intermediate values into registers. Here we assume the combinational path, between any two registers, consists of no more than two adders or one multiplier. Furthermore, parallelism should be exploited as much as possible to avoid unnecessary latency. For instance in statement 1, `(grad - lbd)` and `(param - rmean)` can be done in parallel.

```
# only core part here, see the previous list 1 for variable naming

# statement 1
param -= lr * (grad - lbd + alpha * (param - rmean))

# statement 2
lbd -= lbd - alpha * (param - rmean)

```

```
# statement 3
rmean = 0.5*(param + rmean) - ita * lbd
```

Listing 2: FPTT calculation core part

By the rule stated before, we have a baseline scheduling in table 10. The updated parameter, lambda and running mean are produced at position (P), (L) and (R). The entire latency is 8 cycles, and the throughput is one over 8 cycles. This scheduling can be realized by a finite state machine of 8 states. Note that an extra constant `lr_a` is created by the multiplication of two constant `lr` and `alpha` so that `lr * (grad-lbd)` and `lr_a * (param-rmean)` can both be done at cycle 2. This can reduce the latency of statement 1 by two cycles.

Table 10: baseline scheduling of FPTT logic on hardware

cycle	operations can be done in parallel	
	op1	op2
1	c0_0 = grad – lbd	c0_1 = param – rmean
2	c1_0 = lr * c0_0	c1_1 = lr_a * c0_1
3	(P) c2_0 = parameter – c1_0 – c1_1	
4	c3_0 = c2_0 – rmean	c3_1 = rmean + c2_0
5	c4_0 = alpha * c3_0	c4_1 = 0.5 * c3_1
6	(L) c5_0 = lbd – c4_0	
7		c6_1 = ita * c5_0
8		(R) c7_1 = c4_1 – c6_1

With this naive scheduling ready, two common optimizations can be proposed in terms of throughput and area. For one thing, the throughput can be improved to 1 output per cycle by pipelining, namely saving all intermediate results in registers (table 11). This can make the throughput 8 times faster at the cost of the triple number of registers.

Table 11: pipeline scheduling of FPTT logic on hardware

cycle	operations done in parallel				
	op1	op2	op3	op4	op5
1	c0_reg0 = grad - lbd	c0_reg1 = param – rmean	c0_param = param	c0_lbd = lbd	c0_rm = rmean
2	c1_reg0 = lr * c0_reg0	c1_reg1 = lr_a * c0_reg1	c1_param = c0_param	c1_lbd = c0_lbd	c1_rm = c0_rm
3	c2_reg0 = c1_param – c0_reg0 – c0_reg1			c2_lbd = c1_lbd	c2_rm = c1_rm
4	c3_reg0 = c2_reg0 – c2_rm	c3_reg1 = c2_reg0 + c2_rm	c3_out_p = c2_reg0	c3_lbd = c2_lbd	
5	c4_reg0 = alpha * c3_reg0	c4_reg1 = 0.5 * c3_reg1	c4_out_p = c3_out_p	c4_lbd = c3_lbd	
6	c5_reg0 = c4_lbd – c4_reg0	c5_reg1 = c4_reg1	c5_out_p = c4_out_p		
7	c6_reg0 = ita * c5_reg0	c6_reg1 = c5_reg1	c6_out_p = c5_out_p	c6_out_l = c5_reg0	
8	(R) c7_reg0 = c6_reg1 - c6_reg0		(P) c7_out_p = c6_out_p	(L) c7_out_l = c6_out_l	

For another, an area-compact design can be created from the perspective of the computational entity, like a multiplier or adder. In naive scheduling, each cycle corresponds to one state of the FSM. It can be observed that the multiplier used in cycle/state 2 is only busy in that cycle, and so does the multiplier used in cycle/state 5. In other words, multiplications in cycle 2 and 5 can be mapped on the same multiplier. In such a way, the area can be saved by removing many unnecessary computational entities, at the cost of relatively smaller multiplexers for switching the input of the entity (table 12). Note that registers are also reused as much as possible, while `T1` and `T2` are two general-purpose registers.

Table 12: area-compact scheduling of FPTT logic on hardware

cycle	operations mapped on the entity	
	Subtractor (Adder)	Multiplier
1	T1 = grad – lbd	
2	T2 = param – rmean	T1 = T1 * lr
3	T1 = param – T1	T2 = T2 * lr_a
4	(P) param = T1 – T2	
5	T2 = param – rmean	
6	T1 = param – (- rmean)	T2 = T2 * a
7	(L) lbd = lbd – T2	T1 = T1 * 0.5
8		T2 = lbd * ita
9	(R) rmean = T1 – T2	

6.3 Basic study of general-purpose architectures

As decided before, we choose to implement network forward and backward passes on a general-purpose architecture. So it is necessary to investigate common options. Since forward and backward passes are the bottleneck of the routine to be optimized in terms of computations, we hope the architecture can be optimized and customized to compensate for the inefficiency brought by flexibility. So it is ideal to use an architecture that can support customization towards parallelism, i.e. efficient vector and matrix operations. Due to the time limit, this study is still ongoing. For now, we basically introduce the common choices and in-depth study and decision-making are to be done in the next steps.

Transport-triggered architecture (TTA) [10] is a kind of processor design in which programs directly control the internal transport buses of a processor. Computation happens as a side effect of data transports: writing data into a triggering port of a functional unit triggers the functional unit to start a computation. This is similar to what happens in a systolic array. Due to its modular structure, TTA is an ideal processor template for application-specific instruction set processors (ASIP) with customized datapaths but without the inflexibility and design cost of fixed function hardware accelerators.

RISC-V [11] is an open standard instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles. Unlike most other ISA designs, RISC-V is provided under royalty-free open-source licenses. A number of companies are offering or have announced RISC-V hardware; open source operating systems with RISC-V support are available, and the instruction set is supported in several popular software toolchains.

6.4 Summary

In this section, the hardware architecture is proposed based on the developed Python program. Then it is quantitatively analyzed in terms of the memory requirement, the amount of computation and the execution time. It can be observed that network forward pass and backward pass are the bottleneck or foundation of the system to be optimized, while the load of FPTT logic is relatively trivial. We decided to implement the system with a general-purpose architecture having the possibility to be optimized for vector or matrix operations.

On the other hand, we propose a subsystem for FPTT logic according to its parallelism in calculation. Also, a fixed data path for FPTT logic is considered, as well as two general optimizations in terms of throughput and area.

7 Conclusion

In the preparation phase, we first studied FPTT theory and required background knowledge. Then we explore the theory towards the hardware implementation, during which we developed a Python program of FPTT training on a simplified network configuration using NumPy, i.e. computations are transparent without PyTorch. Last, we proposed the hardware architecture and analyzed its bottleneck, which directs the future implementation.

8 Next steps and planning

Table 13: Timeline of next steps

Time/Weeks	Task	Deliverables
2	in-depth study of potential general-purpose architectures	document
4	Logic design of the scalable system	RTL files
2	Design space exploration	document
3	Physical design of the system	netlist
2	Performance evaluation and comparison	document
3	Final thesis writing and presentation preparation	report and slides

References

[1] H. Cho, J. Lee, and J. Lee. "FARNN: FPGA-GPU Hybrid Acceleration Platform for Recurrent Neural Networks". In: *IEEE Transactions on Parallel & Distributed Systems* 33.07 (July 2022), pp. 1725–1738. ISSN: 1558-2183. DOI: 10.1109/TPDS.2021.3124125.

[2] Anil Kag and Venkatesh Saligrama. "Training Recurrent Neural Networks via Forward Propagation Through Time". In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, 18–24 Jul 2021, pp. 5189–5200. URL: https://proceedings.mlr.press/v139/kag21a.html.

[3] Ji Lin et al. *On-Device Training Under 256KB Memory*. 2022. arXiv: 2206.15472 [cs.CV].

[4] Wikipedia contributors. *Tensor Processing Unit — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Tensor_Processing_Unit&oldid=1172592067. [Online; accessed 30-August-2023]. 2023.

[5] Stefano Fusi et al. "Spike-Driven Synaptic Plasticity: Theory, Simulation, VLSI Implementation". In: *Neural Computation* 12.10 (Oct. 2000), pp. 2227–2258. ISSN: 0899-7667. DOI: 10.1162/089976600300014917. eprint: https://direct.mit.edu/neco/article-pdf/12/10/2227/814631/089976600300014917.pdf. URL: https://doi.org/10.1162/089976600300014917.

[6] J. Sjöström and W. Gerstner. "Spike-timing dependent plasticity". In: *Scholarpedia* 5.2 (2010). revision #184913, p. 1362. DOI: 10.4249/scholarpedia.1362.

[7] Guillaume Bellec et al. "A solution to the learning dilemma for recurrent networks of spiking neurons". In: *Nature Communications* 11.1 (July 2020), p. 3625. ISSN: 2041-1723. DOI: 10.1038/s41467-020-17236-y. URL: https://doi.org/10.1038/s41467-020-17236-y.

[8] Bojian Yin, Federico Corradi, and Sander M. Bohté. "Accurate online training of dynamical spiking neural networks through Forward Propagation Through Time". In: *CoRR* abs/2112.11231 (2021). arXiv: 2112.11231. URL: https://arxiv.org/abs/2112.11231.

[9] Wikipedia contributors. *Long short-term memory — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Long_short-term_memory&oldid=1171845231. [Online; accessed 28-August-2023]. 2023.

[10] Wikipedia contributors. *Transport triggered architecture — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Transport_triggered_architecture&oldid=1127575124. [Online; accessed 30-August-2023]. 2022.

[11] Wikipedia contributors. *RISC-V — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=RISC-V&oldid=1170959711. [Online; accessed 30-August-2023]. 2023.