Master Thesis
Neural Information Processing

BackPACK for Residual and Recurrent Neural Networks

Graduate Training Centre of Neuroscience

Faculty of Science
Faculty of Medicine

Eberhard Karls Universität Tübingen

Tim Schäfer

from Geislingen a.d. Steige, Germany

Tübingen, September 30, 2021

Thesis Advisor:	Prof. Dr. Philipp Hennig Methods of Machine Learning Department of Computer Science
Second Reader:	Prof. Dr. Zeynep Akata Explainable Machine Learning Cluster of Excellence – Machine Learning for Science

Disclosures:

- I affirm that I have written the thesis myself and have not used any sources and aids other than those indicated.
- I affirm that I have not included data generated in one of my laboratory rotations and already presented in the respective laboratory report.

City, Date   Signature

Abstract

Neural networks are trained and analyzed with autodifferentiation engines like PyTorch. These frameworks are limited to computing mini-batch gradients. This restricts training algorithms as well as analysis for deep neural networks. BackPACK offers an efficient and convenient implementation for computing a set of additional quantities. However, residual and recurrent neural networks are not supported. Expanding BackPACK to these architectures is the main goal of this thesis. For this purpose, the required Jacobian-vector products are calculated, the derived algorithms are implemented in BackPACK, and the framework is adjusted to the new architectures. As a result, all torchvision ResNets and a range of RNNs are supported. The benchmarks show that first and second order quantities can be computed efficiently. Other projects like Laplace and Cockpit will directly benefit from BackPACK's improvements. This enables more researchers to gain new insights and develop new analyzing methods or training algorithms.

Acknowledgements

I would like to thank Felix Dangel, who offered plenty of support. On a weekly basis, he helped me to set goals and reach new levels of coding quality. I enjoyed our discussions on theory and software design.

I would also like to thank my supervisor, Professor Philipp Hennig, who guided this project with his expertise.

Finally, I would like to thank the whole Methods of Machine Learning group for offering an awesome working environment. This includes both letting me use their computers and also a rich social life. I enjoyed the conversations during lunch and coffee/tea breaks.

Contents

Chapter 1

Introduction

1.1 Background on Autodifferentiation

Autodifferentiation engines help with the optimization of deep learning architectures. For this task, engines like PyTorch (Paszke et al., 2019) and Tensorflow (Abadi et al., 2015) grant efficient access to the gradient. An inherent issue of the optimized gradient computation is the limitation this means for the optimization algorithms. The most popular algorithms are iterative first order methods (Goodfellow et al., 2016, p. 177) (Schneider et al., 2019). These are for example SGD (Robbins and Monro, 1951), Momentum (Polyak, 1964; Nesterov, 1983; Rumelhart et al., 1986), and Adam (Kingma and Ba, 2015). Alternative approaches like second order methods are always hindered by this lack of easy and efficient access to alternative quantities in standard libraries. This problem can be overcome with BackPACK (Dangel et al., 2019). It is an extension for PyTorch, that grants efficient access to additional quantities. Projects like Cockpit (Schneider et al., 2021), and Laplace (Daxberger et al., 2021) already use BackPACK. However, BackPACK does not support many architectures. Especially, popular architectures like ResNets (He et al., 2016), and RNNs (Rumelhart et al., 1986), including LSTMs (Hochreiter and Schmidhuber, 1997; Sak et al., 2014) are not supported. This master thesis has the goal of making these two architectures available in BackPACK. As a result, related projects using BackPACK automatically expand their support to these architectures as well. Furthermore, it might motivate other researchers (within and outside of the group) to try out BackPACK and related projects.

1.2 Thesis Structure

Section 1.3 gives some insights of how BackPACK works and how it interacts with PyTorch.

Chapter 2 lists the code changes in BackPACK. This chapter's purpose is not only to show which theoretical considerations have been necessary, but also to be a reference in case the code is unclear or a similar module should be included in BackPACK. In the first part, section 2.1, it is explained which modules of ResNets have been missing in BackPACK and how these have been implemented. In the second part, section 2.2, we explain which RNN modules are now included into BackPACK and which additional modules have been created to support RNNs in BackPACK. For RNNs, this is the major part. In section 2.3, other code contributions are listed, such as the creation of tutorials, the update of the backward hook to the latest PyTorch versions, and the allowance of the option `retain_graph`.

Chapter 3 introduces the converter function that helps users to conveniently use the new functionality with small code additions. For each of the architectures, it is explained which exact conversions are performed. For ResNets, explained in section 3.1, the handling of the branching computation graph is explained. For ResNets, this is the major point of improvement in this thesis that allows for a large variety of architectures. For RNNs, explained in section 3.2, it is explained what the converter function does to conveniently use BackPACK together with ResNets.

In chapter 4 we sum up the project's results. Section 4.1 explains which architectures are now supported in BackPACK. This support consists of three parts. Firstly, we give an overview over which single modules have BackPACK support. Secondly, which kind of RNN architectures, and thirdly, which kind of ResNet architectures are now supported. After that, in section 4.2, benchmarks for the new architectures are presented to show that the new architectures are efficiently accessible. The benchmarks consist of the setup and results for Tolstoi Char RNN and a Wide ResNet. Section 4.3 gives a code example. It shows that the code needed to use the new architectures with the converter is minimal.

Finally, chapter 5 concludes with a discussion of the improvements that have been made. Additionally, it lays out future applications for BackPACK.

1.3 A Brief Introduction to BackPACK

The basis of BackPACK is the PyTorch framework, where BackPACK is built on top of it. PyTorch computes the basic gradients and if BackPACK is used, several additional computations are performed. The explanations in this section how BackPACK interacts with PyTorch are based on Dangel et al. (2019) and use the same notation and figures. For more details consult Dangel et al. (2019).

Notation and forward pass Here, we assume that the whole neural network is a concatenation of modules, indexed with i. The module i has input $\{\boldsymbol{z}_n^{(i-1)}\}_{n=1}^N$ and output $\{\boldsymbol{z}_n^{(i)}\}_{n=1}^N$, with the number of samples N. The module transforms the input with the transformation $T_{\boldsymbol{\theta}^{(i)}}^{(i)}$, where $\boldsymbol{\theta}^{(i)}$ denotes the module's parameters.

Standard gradient computation First, we look at how PyTorch computes the gradients in Figure 1.1. During the backward pass, the module receives the gradient from its succeeding module. Then, two different Jacobians are applied. On the one hand, the Jacobian wrt parameters is applied to the backpropagated gradients and the result is saved as the parameter's gradient $\nabla_{\boldsymbol{\theta}^{(i)}}\mathcal{L}$. On the other hand, the Jacobian wrt module's input is applied to the backpropagated gradients and the result is passed to the preceding module. Note, that the gradient wrt the parameters is summed over the batch axis. Standard training algorithms like Adam use this gradient to perform some form of iterative step on the parameter and then delete it.

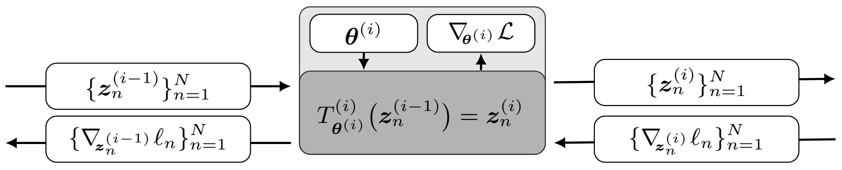

$\boldsymbol{\theta}^{(i)}$
$\nabla_{\boldsymbol{\theta}^{(i)}}\mathcal{L}$
$\{\boldsymbol{z}_n^{(i-1)}\}_{n=1}^N$
$\{\nabla_{\boldsymbol{z}_n^{(i-1)}}\ell_n\}_{n=1}^N$
$T_{\boldsymbol{\theta}^{(i)}}^{(i)}(\boldsymbol{z}_n^{(i-1)}) = \boldsymbol{z}_n^{(i)}$
$\{\boldsymbol{z}_n^{(i)}\}_{n=1}^N$
$\{\nabla_{\boldsymbol{z}_n^{(i)}}\ell_n\}_{n=1}^N$

Figure 1.1: Gradient computation in PyTorch for module i with N samples (Dangel et al., 2019, Fig. 2)

Hooks as interface To compute additional quantities BackPACK makes use of an interface provided by PyTorch. PyTorch allows for forward and backward hooks to be installed on modules. These hooks are executed right after the respective PyTorch routine. In the forward hook, BackPACK saves quantities for later, like input and output. In the backward hook, BackPACK uses these quantities to compute additional things like individual gradients. Note that in this master thesis, the interface that is used by BackPACK changed from `register_backward_hook` to `register_full_backward_hook`.

First order extensions For first order quantities like individual gradients (illustrated in Figure 1.2), the additional operations are fairly simple. For example, the BatchGrad extension (computing individual gradients) applies the Jacobian wrt parameters onto the backpropagated gradients. This yields the individual gradients $\{\nabla_{\boldsymbol{\theta}^{(i)}} \ell_n\}_{n=1}^N$. If one sums these quantities along the batch axis one receives the gradient $\nabla_{\boldsymbol{\theta}^{(i)}} \mathcal{L}$. For first order extensions, each of these module extensions acts independently. Therefore, this works even if not all of the network's modules are supported.

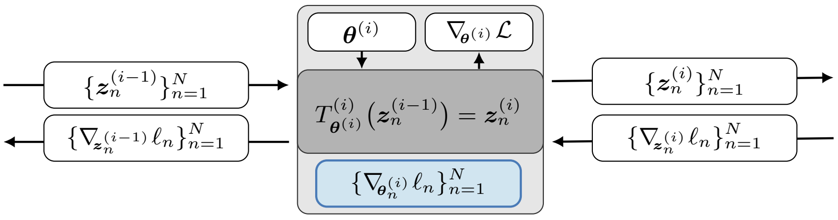

$\boldsymbol{\theta}^{(i)}$
$\nabla_{\boldsymbol{\theta}^{(i)}} \mathcal{L}$
$\{\boldsymbol{z}_n^{(i-1)}\}_{n=1}^N$
$\{\nabla_{\boldsymbol{z}_n^{(i-1)}} \ell_n\}_{n=1}^N$
$T_{\boldsymbol{\theta}^{(i)}}^{(i)}(\boldsymbol{z}_n^{(i-1)}) = \boldsymbol{z}_n^{(i)}$
$\{\boldsymbol{z}_n^{(i)}\}_{n=1}^N$
$\{\nabla_{\boldsymbol{z}_n^{(i)}} \ell_n\}_{n=1}^N$
$\{\nabla_{\boldsymbol{\theta}_n^{(i)}} \ell_n\}_{n=1}^N$

Figure 1.2: Individual gradient computation in BackPACK for module i with N samples (Dangel et al., 2019, Fig. 4)

Second order extensions For second order quantities, additional steps are required. Specifically, some quantities need to be passed from module to module, illustrated in Figure 1.3 for the DiagGGN extension. The whole extension is based around the idea that the loss function's Hessian has a factorization $S_n S_n^T = \nabla_f^2 l(f(\boldsymbol{\theta}, \boldsymbol{x}_n), \boldsymbol{y}_n)$. In this case, all subsequent modules can then calculate their GGN quantity using the recursive formula seen in the figure. This requires that this additional information is passed from module to module. Since PyTorch currently does not offer a native method for passing the quantities BackPACK has to implement this itself. This structure implies that every single module in the computation graph must be supported in BackPACK.

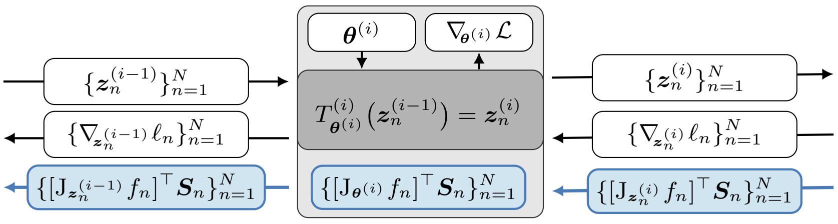

$\boldsymbol{\theta}^{(i)}$
$\nabla_{\boldsymbol{\theta}^{(i)}} \mathcal{L}$
$\{\boldsymbol{z}_n^{(i-1)}\}_{n=1}^N$
$\{\nabla_{\boldsymbol{z}_n^{(i-1)}} \ell_n\}_{n=1}^N$
$T_{\boldsymbol{\theta}^{(i)}}^{(i)}(\boldsymbol{z}_n^{(i-1)}) = \boldsymbol{z}_n^{(i)}$
$\{\boldsymbol{z}_n^{(i)}\}_{n=1}^N$
$\{\nabla_{\boldsymbol{z}_n^{(i)}} \ell_n\}_{n=1}^N$
$\{[\mathrm{J}_{\boldsymbol{z}_n^{(i-1)}} f_n]^\top \boldsymbol{S}_n\}_{n=1}^N$
$\{[\mathrm{J}_{\boldsymbol{\theta}^{(i)}} f_n]^\top \boldsymbol{S}_n\}_{n=1}^N$
$\{[\mathrm{J}_{\boldsymbol{z}_n^{(i)}} f_n]^\top \boldsymbol{S}_n\}_{n=1}^N$

Figure 1.3: Second order quantities computation in BackPACK (Dangel et al., 2019, Fig. 5)

ResNets and RNNs In principle this scheme can be transferred to ResNets and RNNs because they also have this modular structure. However, there are several challenges.

- First, the modules required by the new architecture have to be supported. This involves calculating the derivatives of the modular functions and implementing them. The calculations are presented in section 2.1 and 2.2.

- Secondly, sometimes ResNets and RNNs have different input and output format. How this is handled is explained in section 3.1 for ResNets and section 3.2 for RNNs.

Chapter 2

Additions to BackPACK

A large part of this thesis is code contribution to BACKPACK. It consists of more than 8.000 lines[1]. This chapter lays out the necessary calculations for the code and other contributions that happened during the thesis.

Jacobian-vector products notation First of all, the basic modules that are used in ResNets and RNNs must be added to BACKPACK. To implement these basic modules, the Jacobian vector products must be available for each module. Here, I define the Jacobian as $J_{\boldsymbol{a}}\boldsymbol{y}$ as $[J_{\boldsymbol{a}}\boldsymbol{y}]_{ij} = \frac{\partial \boldsymbol{y}_i}{\partial \boldsymbol{a}_j}$. As the goal is to compute Jacobian-vector products, we define a short notation for a product up to node $\boldsymbol{a}$ with the vector M that should be multiplied: $A = (J_{\boldsymbol{a}}\boldsymbol{y})^T M$. This notation has the convenient property that subsequent calculations can reuse the product that is already calculated using the chain rule:

$$X = (J_{\boldsymbol{x}}\boldsymbol{y})^T M = (J_{\boldsymbol{x}}\boldsymbol{a})^T (J_{\boldsymbol{a}}\boldsymbol{y})^T M = (J_{\boldsymbol{x}}\boldsymbol{a})^T A.$$

Shapes Assume that the module has input $\boldsymbol{x}$ and output $\boldsymbol{y}$ with respective shapes `shape_in` and `shape_out`. Then the transposed Jacobian wrt the input $(J_{\boldsymbol{x}}\boldsymbol{y})^T$ has shape $(\texttt{shape\_in}, \texttt{shape\_out})$. The vector/matrix M has an additional free axis V to allow for efficient vectorization. So its shape is $(V, \texttt{shape\_out})$. The resulting product $X = (J_{\boldsymbol{x}}\boldsymbol{y})^T M$ has shape $(V, \texttt{shape\_in})$.

Required Jacobian-vector products

- Transposed Jacobian wrt parameters $P_{\boldsymbol{\theta}} = (J_{\boldsymbol{\theta}}\boldsymbol{y})^T M$: This is the basic Jacobian that must be implemented for first order extensions, like individual gradients.

[1]An exact count will be available in the github metrics with the next BACKPACK release. This is a rough approximation summing up code lines in pull requests. This line count includes documentation, examples, and tests.

- Transposed Jacobian wrt input $X = (J_{\boldsymbol{x}}\boldsymbol{y})^T M$: This is the product that is necessary for second order extensions, like the DiagGGN. It is used to pass quantities back to the preceding module.
- Jacobian wrt input $P = M(J_{\boldsymbol{x}}\boldsymbol{y})$ (with shape of M: $(V, \texttt{shape\_in})$): This product is required for some special second order extensions, like GGNMP.

2.1 Complementing Modules for ResNets

2.1.1 nn.BatchNorm

Before: only training mode, only 1d
After: both evaluation and training mode; 1d, 2d, and 3d
Code: PR 179 and PR 201

Forward pass in training mode The BatchNorm module can improve training performance (Ioffe and Szegedy, 2015). It centralizes the batch data using mean and variance of a dataset. During training, additionally to rescaling the data, the statistics are learned and saved for evaluation. During evaluation, these learned statistics are used for rescaling instead of the batch statistics. The formula for training mode is (PyTorch Documentation BatchNorm2d)

$$\boldsymbol{y} = \frac{\boldsymbol{x} - \mathrm{E}[\boldsymbol{x}]}{\sqrt{\mathrm{Var}[\boldsymbol{x}] + \epsilon}} * \boldsymbol{\gamma} + \boldsymbol{\beta}$$

with bias $\boldsymbol{\beta}$, weight $\boldsymbol{\gamma}$, and numerical stabalizer ϵ.

The details of calculations for training mode are explained in Kevin Zakka's blog, in Chris Yeh's blog, and in Paul Fischer's research project report.

Evaluation mode In the previously not supported evaluation mode, because the statistics don't depend on the data anymore, we rename them to $\mathrm{E}[\boldsymbol{x}] = \boldsymbol{\mu}$ and $\mathrm{Var}[\boldsymbol{x}] = \boldsymbol{\sigma}^2$. In the Jacobians, we can drop all terms that stem from this dependency.

Transposed Jacobian wrt parameters is the Jacobian wrt $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$:

$$\begin{aligned}(J_{\boldsymbol{\gamma}}\boldsymbol{y})^T &= \frac{\boldsymbol{x} - \boldsymbol{\mu}}{\sqrt{\boldsymbol{\sigma}^2 + \epsilon}} \\ (J_{\boldsymbol{\beta}}\boldsymbol{y})^T &= \mathbb{I}\end{aligned}$$

Transposed Jacobian wrt input

$$(J_{\boldsymbol{x}}\boldsymbol{y})^T = \frac{\boldsymbol{\gamma}}{\sqrt{\boldsymbol{\sigma}^2 + \epsilon}}$$

2.1.2 nn.AdaptiveAvgPool

Before: no support for AdaptiveAvgPool[123]d
After: support for AdaptiveAvgPool[123]d if equivalent AvgPool[123]d exists
Code: PR 165 and PR 201

The AdaptiveAvgPool module (PyTorch Documentation AdaptiveAvgPool) is a convenient version of the AvgPool module. The user can simply define a target size instead of `stride`, `kernel_size`, and `padding`. In torchvision ResNets, it is only used to reduce the output to size (1,1). In this case, there is an equivalent AvgPool module that does the same.

Equivalent AvgPool module More generally, it exists an equivalent AvgPool module if in each dimension, the original dimension size is a multiple of the target's corresponding dimension size. The equivalent parameters in each dimension are (stackoverflow post)

$$\begin{aligned} \texttt{stride} &= (\texttt{input\_size}//\texttt{output\_size}) \\ \texttt{kernel\_size} &= \texttt{input\_size} - (\texttt{output\_size} - 1) * \texttt{stride} \\ \texttt{padding} &= 0 \end{aligned}$$

It this way, for a lot of AdaptiveAvgPool modules, an equivalent AvgPool module is determined and the derivatives are recycled.

Bug in AdaptiveAvgPool3d On a sidenote, since BackPACK tests its derivatives and compares them to the PyTorch version, we found a bug in PyTorch for AdaptiveAvgPool3d on CUDA. We reported this in the PyTorch forum which led to a github issue and the PyTorch team quickly fixed it in PyTorch pull request 60630. This fix will be available in future releases.

2.1.3 SumModule

Before: no such module
After: custom module that does the same as built-in function `add`
Code: PR 202

In ResNets, because of their structure the built-in function `add` is repeatedly used. However, for second order extensions, BackPACK requires that all operations must be modules. Therefore, the custom module SumModule is introduced. It does just the same as a summation. Therefore, the derivatives are straightforward: Identity in all cases.

2.2 Complementing Modules for RNNs

2.2.1 nn.RNN

Before: no support
After: supported
Code: PR 156, PR 158, and PR 159

Forward pass The forward pass of the RNN module is (PyTorch documentation RNN)

$$\boldsymbol{h}_t = \tanh(W_{ih}\boldsymbol{x}_t + \boldsymbol{b}_{ih} + W_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{hh}).$$

We rewrite this equation with parameter names $V = W_{ih}$, $W = W_{hh}$, $\boldsymbol{b} = \boldsymbol{b}_{ih}$, and $\boldsymbol{c} = \boldsymbol{b}_{hh}$

$$\begin{aligned}\boldsymbol{a}_t &= V\boldsymbol{x}_t + \boldsymbol{b} + W\boldsymbol{h}_{t-1} + \boldsymbol{c}\\ \boldsymbol{h}_t &= \tanh(\boldsymbol{a}_t).\end{aligned}$$

This computation is visualized in Figure 2.1.

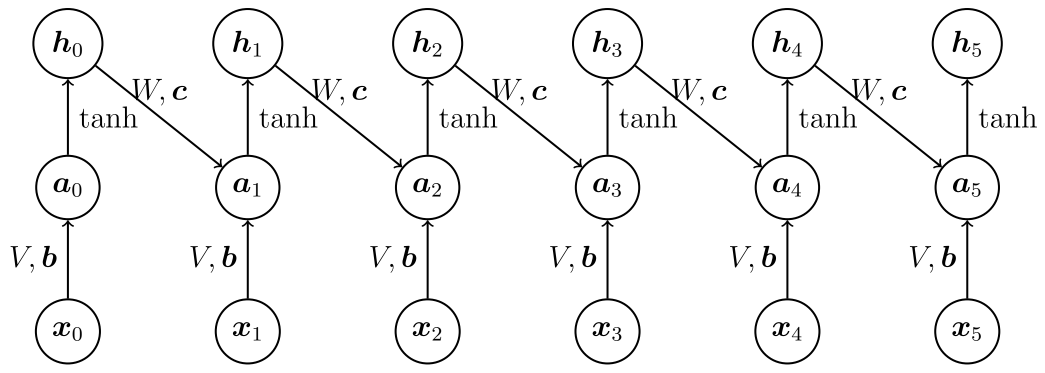


Figure 2.1: Computation graph of RNN module with six time steps.

Jacobian vector products and shapes The input $\boldsymbol{x}$ has shape $(N, T,$ `input_size`$)$. All hidden variables $\boldsymbol{a}$ and $\boldsymbol{h}$ have shape $(N, T,$ `hidden_size`$)$. Now, we derive the Jacobian-vector products listed at the beginning of chapter 2. Goodfellow et al. (2016, pp. 384-6) compute the gradient of the loss wrt these quantities and was used as a first orientation for deriving the equations here. We calculate the product $X = (J_{\boldsymbol{x}}\boldsymbol{h})^T M$. It is calculated by passing partial products of the Jacobian and the matrix M backwards. At the last time step $t = T$ only the matrix M is passed backwards. For all other time steps t, additionally, the quantities of the successive time steps are considered.

Jacobian-vector product up to node $\boldsymbol{a}$ Here, we formulate the backward pass wrt $\boldsymbol{a}_t$. It can be formulated as a backward recursion:

$$\begin{aligned} A_T &= (J_{\boldsymbol{a}_T}\boldsymbol{h}_T)^T M_T && (2.1a)\\ A_t &= (J_{\boldsymbol{a}_t}\boldsymbol{h}_t)^T \big(M_t + (J_{\boldsymbol{h}_t}\boldsymbol{a}_{t+1})^T A_{t+1}\big) \qquad \text{for } t = T-1, ..., 0. && (2.1b)\end{aligned}$$

In this formula, the required quantities are

$$\begin{aligned} (J_{\boldsymbol{a}_t}\boldsymbol{h}_t)^T &= \operatorname{diag}(1 - \boldsymbol{h}_t^2)\\ (J_{\boldsymbol{h}_t}\boldsymbol{a}_{t+1})^T &= W^T. \end{aligned}$$

Inserting this into equation 2.1 yields

$$\begin{aligned} A_T &= \operatorname{diag}(1 - \boldsymbol{h}_T^2) M_T\\ A_t &= \operatorname{diag}(1 - \boldsymbol{h}_t^2)(M_t + W^T A_{t+1}) \qquad \text{for } t = T-1, ..., 0. \end{aligned}$$

Transposed Jacobian wrt input The Jacobian of $\boldsymbol{a}_t$ wrt the input is

$$(J_{\boldsymbol{x}_r}\boldsymbol{a}_t)^T = V^T \delta_{tr}.$$

Therefore, the Jacobian matrix product wrt the input is

$$X_t = \sum_r (J_{\boldsymbol{x}_r}\boldsymbol{a}_t)^T A_r = \sum_r V^T \delta_{tr} A_r = V^T A_t$$

Transposed Jacobian wrt parameters The Jacobians of $\boldsymbol{a}_t$ wrt the parameters are

$$\begin{aligned} (J_{\boldsymbol{b}}\boldsymbol{a}_t)^T &= \mathbb{I}\\ (J_V\boldsymbol{a}_t)^T &= \boldsymbol{x}_t\\ (J_W\boldsymbol{a}_t)^T &= \boldsymbol{h}_{t-1} \end{aligned}$$

and the product is then

$$\begin{aligned} P_b &= \sum_t A_t\\ P_V &= \sum_t \boldsymbol{x}_t A_t\\ P_W &= \sum_t \boldsymbol{h}_{t-1} A_t \end{aligned}$$

Jacobian wrt input For the product $P = M(J_{\boldsymbol{x}}\boldsymbol{h})$ of M with the Jacobian, the order of computation is reversed. We first compute the Jacobian vector product up to node $\boldsymbol{a}_t$.

$$A_t = M_t(J_{\boldsymbol{x}_t}\boldsymbol{a}_t)$$

Afterwards, we calculate the product with the Jacobian up to node $\boldsymbol{h}$ recursively.

$$\begin{aligned} P_0 &= A_0(J_{\boldsymbol{a}_0}\boldsymbol{h}_0) \\ P_t &= \big(A_t + P_{t-1}(J_{\boldsymbol{h}_{t-1}}\boldsymbol{a}_t)\big)(J_{\boldsymbol{a}_t}\boldsymbol{h}_t) \qquad \text{for } t = 1, ..., T \end{aligned}$$

Filling in the Jacobians yields the recursive formula to implement:

$$\begin{aligned} P_0 &= M_0 V \operatorname{diag}(1 - \boldsymbol{h}_0^2) \\ P_t &= (M_t V + P_{t-1} W) \operatorname{diag}(1 - \boldsymbol{h}_t^2) \qquad \text{for } t = 1, ..., T \end{aligned}$$

2.2.2 nn.LSTM

Before: no support
After: supported
Code: PR 169 and PR 215

Forward pass The forward pass of the LSTM layer is defined as (PyTorch documentation LSTM):

$$\begin{aligned} \widetilde{\boldsymbol{ifgo}}_t &= W_{ih}\boldsymbol{x}_t + \boldsymbol{b}_{ih} + W_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{hh} \\ \boldsymbol{ifgo}_t &= \big(\sigma(\tilde{\boldsymbol{i}}_t) \quad \sigma(\tilde{\boldsymbol{f}}_t) \quad \tanh(\tilde{\boldsymbol{g}}_t) \quad \sigma(\tilde{\boldsymbol{o}}_t)\big) \\ \boldsymbol{c}_t &= \boldsymbol{f}_t \cdot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \cdot \boldsymbol{g}_t \\ \boldsymbol{h}_t &= \boldsymbol{o}_t \cdot \tanh(\boldsymbol{c}_t) \end{aligned}$$

Here, the shapes are similar to the RNN, but there are more hidden variables. The hidden variables $\boldsymbol{i}$, $\boldsymbol{f}$, $\boldsymbol{g}$, and $\boldsymbol{o}$ are sometimes handled as a single variable $\boldsymbol{ifgo}$ for shorter notation. The computation is also visualized in Figure 2.2.

Transposed Jacobian matrix product until $\widetilde{\boldsymbol{ifgo}}$ First of all, we compute the product up to the nodes $\widetilde{\boldsymbol{ifgo}}_t$. From this point on, the Jacobian wrt the parameters and the inputs can be computed. For this purpose we list all the products that must be computed to reach this step.

For each time step, we compute the product H_t:

$$\begin{aligned} H_T &= M_T \\ H_t &= M_t + (J_{\boldsymbol{h}_t}\widetilde{\boldsymbol{ifgo}}_{t+1})^T \widetilde{IFGO}_{t+1} \qquad \text{for } t = T-1, ..., 0 \end{aligned}$$

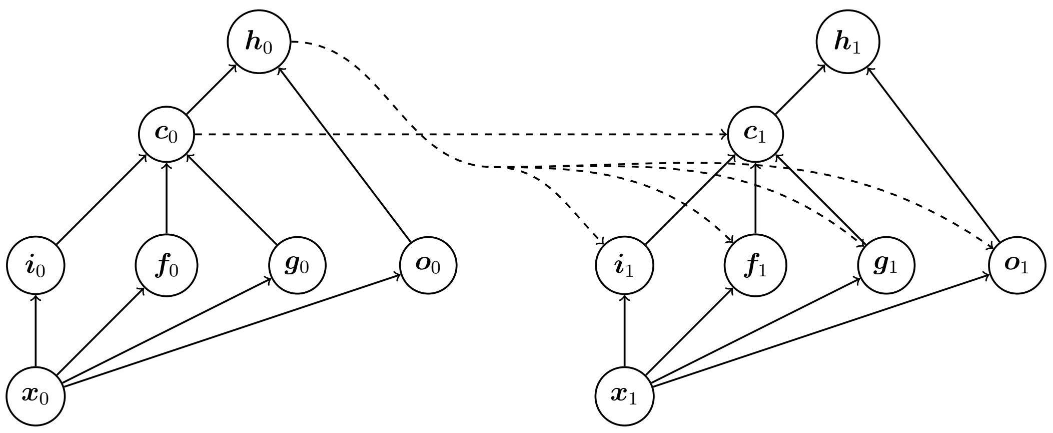


Figure 2.2: LSTM computation graph with two time steps.

Additionally, for each time step, we compute the product C_t:

$$\begin{aligned} C_T &= (J_{\boldsymbol{c}_T}\boldsymbol{h}_T)^T H_T \\ C_t &= (J_{\boldsymbol{c}_t}\boldsymbol{h}_t)^T H_t + (J_{\boldsymbol{c}_t}\boldsymbol{c}_{t+1})^T C_{t+1} \qquad \text{for } t = T-1, ..., 0 \end{aligned}$$

Furthermore, for each time step, we compute the product for the hidden variables $IFGO_t$:

$$\begin{aligned} I_t &= (J_{\boldsymbol{i}_t}\boldsymbol{c}_t)^T C_t \\ F_t &= (J_{\boldsymbol{f}_t}\boldsymbol{c}_t)^T C_t \\ G_t &= (J_{\boldsymbol{g}_t}\boldsymbol{c}_t)^T C_t \\ O_t &= (J_{\boldsymbol{o}_t}\boldsymbol{c}_t)^T H_t \end{aligned}$$

And finally, for each time step, the product $\widetilde{IFGO}_t$:

$$\widetilde{IFGO}_t = (J_{\widetilde{\boldsymbol{ifgo}}_t}\boldsymbol{ifgo}_t)^T IFGO_t.$$

Now, all the intermediate products are defined. The Jacobians that are required to compute these products are:

$$\begin{aligned} (J_{\boldsymbol{h}_t}\widetilde{\boldsymbol{ifgo}}_{t+1})^T &= W_{hh}^T \\ (J_{\boldsymbol{c}_t}\boldsymbol{h}_t)^T &= \boldsymbol{o}_t \cdot (1 - \tanh(\boldsymbol{c}_t)^2) \\ (J_{\boldsymbol{c}_t}\boldsymbol{c}_{t+1})^T &= \boldsymbol{f}_t \\ (J_{\boldsymbol{i}_t}\boldsymbol{c}_t)^T &= \boldsymbol{g}_t \\ (J_{\boldsymbol{f}_t}\boldsymbol{c}_t)^T &= \boldsymbol{c}_{t-1} \\ (J_{\boldsymbol{g}_t}\boldsymbol{c}_t)^T &= \boldsymbol{i}_t \\ (J_{\boldsymbol{o}_t}\boldsymbol{c}_t)^T &= \tanh(\boldsymbol{c}_t) \\ (J_{\widetilde{\boldsymbol{ifgo}}_t}\boldsymbol{ifgo}_t)^T &= \big(\boldsymbol{i}_t(1-\boldsymbol{i}_t) \quad \boldsymbol{f}_t(1-\boldsymbol{f}_t) \quad 1-\boldsymbol{g}_t^2 \quad \boldsymbol{o}_t(1-\boldsymbol{o}_t)\big) \end{aligned}$$

Plugging it all together gives an algorithm for computing the Jacobian matrix product up to node $\widetilde{IFGO}_t$:

- Last time step:

$$\begin{aligned} H_T &= M_T \\ C_T &= \boldsymbol{o}_T \cdot (1 - \tanh(\boldsymbol{c}_T)^2) H_T \end{aligned}$$

- For all other time steps:

$$\begin{aligned} H_t &= M_t + W_{hh}^T \widetilde{IFGO}_{t+1} \\ C_t &= \boldsymbol{o}_t \cdot (1 - \tanh(\boldsymbol{c}_t)^2) H_t + \boldsymbol{f}_t C_{t+1} \\ I_t &= \boldsymbol{g}_t C_t \\ F_t &= \boldsymbol{c}_{t-1} C_t \\ G_t &= \boldsymbol{i}_t C_t \\ O_t &= \tanh(\boldsymbol{c}_t) H_t \\ \widetilde{IFGO}_t &= \big(\boldsymbol{i}_t(1-\boldsymbol{i}_t) \quad \boldsymbol{f}_t(1-\boldsymbol{f}_t) \quad 1-\boldsymbol{g}_t^2 \quad \boldsymbol{o}_t(1-\boldsymbol{o}_t)\big)\, IFGO_t \end{aligned}$$

Transposed Jacobian wrt input Based on this intermediate result, we can compute X_t

$$X_t = (J_{\boldsymbol{x}_t} \widetilde{\boldsymbol{ifgo}}_t)^T \widetilde{IFGO}_t$$

The required Jacobian is

$$(J_{\boldsymbol{x}_t} \widetilde{\boldsymbol{ifgo}}_t)^T = W_{ih}^T.$$

That gives

$$X_t = W_{ih}^T \widetilde{IFGO}_t.$$

Transposed Jacobian wrt parameters For the parameters W_{ih}, $\boldsymbol{b}_{ih}$, W_{hh}, and $\boldsymbol{b}_{hh}$, the Jacobians are

$$\begin{aligned} (J_{W_{ih}} \widetilde{\boldsymbol{ifgo}}_t)^T &= \boldsymbol{x}_t \\ (J_{\boldsymbol{b}_{ih}} \widetilde{\boldsymbol{ifgo}}_t)^T &= \mathbb{I} \\ (J_{W_{hh}} \widetilde{\boldsymbol{ifgo}}_t)^T &= \boldsymbol{h}_{t-1} \\ (J_{\boldsymbol{b}_{hh}} \widetilde{\boldsymbol{ifgo}}_t)^T &= \mathbb{I}. \end{aligned}$$

Consequently, the final products are

$$
\begin{aligned}
P_{W_{ih}} &= \sum_t \boldsymbol{x}_t \widetilde{IFGO}_t \\
P_{\boldsymbol{b}_{ih}} &= \sum_t \widetilde{IFGO}_t \\
P_{W_{hh}} &= \sum_{t=1}^{T} \boldsymbol{h}_{t-1} \widetilde{IFGO}_t \\
P_{\boldsymbol{b}_{hh}} &= \sum_t \widetilde{IFGO}_t
\end{aligned}
$$

Transposed Jacobian wrt input Similarly to the calculations for the RNN module it is the reversed order of computations. The implementation can be seen in BackPACK.

2.2.3 nn.Embedding

Before: no support
After: supported
Code: PR 216

Forward pass The Embedding module is used to transform discrete input into a continuous input. The forward pass is defined as (PyTorch documentation Embedding)

$$
y_{*,h} = \sum_{s=0}^{S} \delta(x_*, s) W_{s,h}
$$

with weight W, the number of embeddings S, number of embedding dimensions H, and the input dimensions $*$. The term $\delta(x_*, s)$ is one if the discrete input x_* matches the index s and zero otherwise. The input can have arbitrary shape $*$ and is integer in the range from 0 to S. The weight matrix W has shape $(S, \texttt{embedding\_dim})$. Therefore, it maps each discrete input to a continuous vector space, where $\boldsymbol{y}$ has the same shape as the input dimensions plus the embedding dimension.

Because the input is discrete the Jacobian wrt input is not defined. The only derivative to compute is wrt weight.

Transposed Jacobian wrt parameter The only parameter is the weight W. Therefore, we must compute the Jacobian $J_{*,h,t,k} = \frac{\partial y_{*,h}}{\partial W_{t,k}}$.

$$\begin{aligned} J_{*,h,t,k} &= \sum_{s=0}^{S} \delta(x_*, s) \frac{\partial W_{s,h}}{\partial W_{t,k}} \\ &= \delta(x_*, t)\delta_{hk}. \end{aligned}$$

Therefore, the product of a matrix $M_{v,*,h}$ with the Jacobian is

$$\begin{aligned} P_{v,t,k} &= \sum_{*,h} \delta(x_*, t)\delta_{hk} M_{v,*,h} \\ &= \sum_{*} \delta(x_*, t) M_{v,*,k} \end{aligned}$$

2.2.4 nn.CrossEntropyLoss

Before: supports only 2d
After: arbitrary number of additional axes
Code: PR 211

Multidimensional inputs A lot of times, the output of RNNs has shape (N, C, T), with number of samples N, number of categories C, and number of time steps T. To compute the cross entropy loss of this output, BackPACK must allow the module CrossEntropyLoss to have additional axes (like the T-axis in this case). These additional axes are allowed due to the PyTorch documentation on CrossEntropyLoss. Luckily, a reshape does the trick. This is because CrossEntropyLoss treats the batch axis and additional axes similarly: as independent samples. Therefore, those axes can be merged and the old algorithms can be reused.

2.2.5 Permute

Before: function permute
After: custom module Permute
Code: PR 158

Forward pass In the previous section on multidimensional cross entropy loss, we saw that the input must be of shape (N, C, T). However, in reality, the output of the network is of shape (N, T, C). To swap the last two axes we require a Permute module. Its forward pass is

$$\boldsymbol{y} = p(\boldsymbol{x})$$

where $\boldsymbol{y}$ is the output, $\boldsymbol{x}$ is the input, and p is the permutation of axes.

Transposed Jacobian wrt input The product $X = (J_{\boldsymbol{x}}\boldsymbol{y})^T M$ of the Jacobian with the matrix is just the same permutation. Note that the first axis is assumed to be the vectorized axis of the matrix M, so it remains unchanged.

$$X = (M[0], p(M[1:]))$$

Jacobian wrt input However, the product $P = M(J_x\boldsymbol{y})$ with the transposed Jacobian is the inverse permutation.

$$P = (M[0], p^{-1}(M[1:]))$$

2.3 Other Contributions

2.3.1 Tutorials

Custom module example tutorial explains how users can add modules to BackPACK. This comes in handy if users are missing modules and don't want to wait for BackPACK to support those.

Residual networks tutorial explains how to use BackPACK with ResNets. It lays out three approaches. First, it explains that for first order extensions it works out of the box. Secondly, it shows how BackPACK internally uses its custom modules to construct a ResNet. Thirdly, it explains how to use the converter function to have convenient transformation to a BackPACK-compatible network.

2.3.2 Full Backward Hook

Before: use `register_backward_hook` (deprecated since torch 1.8.0)
danger of not working with future PyTorch releases
After: use `register_full_backward_hook`
Code: PR 194

Deprecation of `register_backward_hook` The hook BackPACK used is deprecated since torch 1.8.0 and is going to be deleted. This is an issue because BackPACK relies on hooks, even for first order extensions. Therefore, BackPACK was in immediate danger of not working anymore. Thus, it is important to use the new function instead of the deprecated one.

Replacing backward hook Simply replacing the hook does not work immediately. To make the new hook work, BackPACK must change the way it saves the quantities that are passed backwards. Previously, BackPACK used the input tensors to pass these to the preceding module. However, with the new hook, those pointers are not reliable anymore. Instead, the new hook uses the gradient tensors to pass the computed quantities to the preceding module.

Future `register_full_backward_hook` is used from torch 1.9.0 onwards. We decided against using it from 1.8.0 onwards, because the hook does not trigger reliably in its initial release. We now expect BackPACK to work with future PyTorch releases even if `register_backward_hook` is removed.

2.3.3 Allow retain_graph=True

Before: always `retain_graph=False`
After: `retain_graph option`
Code: PR 217

Some users wish to do several backward passes and therefore require the option `retain_graph=True`. Previously, BackPACK ignored this option and deleted its saved quantities no matter what. This is an unexpected behaviour since users know the option `retain_graph=True` and expect a second backward pass to work. With the addition of this option, they can now do several backward passes.

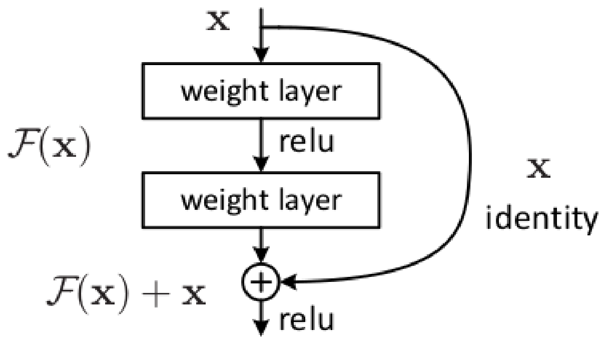
Chapter 3

Converter function: Handling complex architectures

3.1 ResNet: Backpropagation in Graph

Background ResNets have been introduced by He et al. (2016). Since then, these architectures had success in computer vision. For example, Tai et al. (2017) showed that ResNets can achieve superior accuracy with much less parameters on single-image super-resolution compared to large deep neural networks.

Architecture A building block of ResNets can be seen in Figure 3.1. In this block, the computation graph branches out at $\boldsymbol{x}$. In one branch, $\boldsymbol{x}$ goes through a transformation $F(\boldsymbol{x})$. In the other branch the result is kept. At the end of the block, these two branches are summed up.



x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 3.1: Building block of ResNet (He et al., 2016, Fig. 2)

Challenges. ResNets are challenging for BackPACK because they branch out as shown in Figure 3.1. For first order extensions, PyTorch takes care of backpropagating quantities through the branches. However, in second order extensions, BackPACK must define how to propagate quantities through the graph. This yields two major challenges:

- Accumulation function: When the computation graph branches out, this means for the backward pass that the backpropagated quantities have to be accumulated. So far, BackPACK did just overwrite the old result. However, BackPACK must define a correct accumulation function.
- SumModule: The operation that merges these branches again, usually is a summation. This summation is a built-in function and not a PyTorch module. Therefore, BackPACK cannot extend it in a natural way.

Accumulation function Figure 3.2 that PyTorch's internal variables like the gradients are summed up automatically. However, BackPACK's additional quantities must be handled explicitly. In this case, for the DiagGGN extension, the quantities from the different branches are summed up.

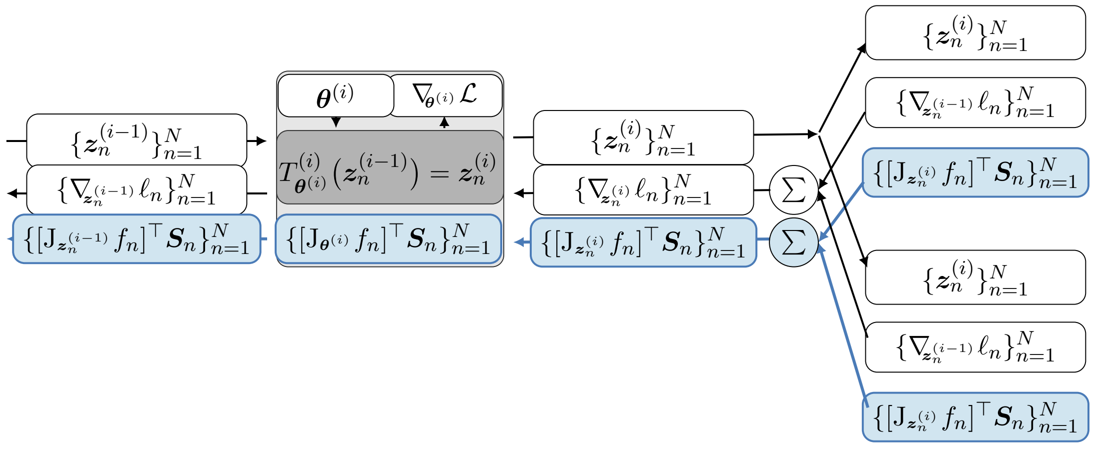

$\{z_n^{(i-1)}\}_{n=1}^N$
$\{\nabla_{z_n^{(i-1)}} \ell_n\}_{n=1}^N$
$\{[J_{z_n^{(i-1)}} f_n]^\top S_n\}_{n=1}^N$
$\theta^{(i)}$
$\nabla_{\theta^{(i)}} \mathcal{L}$
$T_{\theta^{(i)}}^{(i)}(z_n^{(i-1)}) = z_n^{(i)}$
$\{[J_{\theta^{(i)}} f_n]^\top S_n\}_{n=1}^N$
$\{z_n^{(i)}\}_{n=1}^N$
$\{\nabla_{z_n^{(i)}} \ell_n\}_{n=1}^N$
$\{[J_{z_n^{(i)}} f_n]^\top S_n\}_{n=1}^N$
$\sum$
$\sum$
$\{z_n^{(i)}\}_{n=1}^N$
$\{\nabla_{z_n^{(i-1)}} \ell_n\}_{n=1}^N$
$\{[J_{z_n^{(i)}} f_n]^\top S_n\}_{n=1}^N$
$\{z_n^{(i)}\}_{n=1}^N$
$\{\nabla_{z_n^{(i-1)}} \ell_n\}_{n=1}^N$
$\{[J_{z_n^{(i)}} f_n]^\top S_n\}_{n=1}^N$

Figure 3.2: Backpropagation in a ResNet in a ResNet branch.

SumModule The second challenge of not being able to extend the built-in function `add` is solved by the introduction of the custom module SumModule. For this custom module, all the necessary derivatives are defined in BackPACK.

Converter function In principle, implementing these two aspects makes BackPACK support ResNets. However, the resulting interface is not very user-friendly. To achieve BackPACK support, the user must rewrite her

ResNets using the custom modules. This would mean a lot of work, especially for large networks like the ones provided in torchvision. This stands in contrast with the goal of BackPACK to be easy to use. Probably, a lot of researchers would shy away from using BackPACK under such circumstances. This is why a converter function is introduced.

The converter function provides an easy interface to make the model BackPACK-compatible. To use it, the user must set `use_converter=True` in `extend()`. In this case, the computation graph is analyzed with the help of `torch.fx`. Afterwards, the converter function makes several transformations:

- Built-in function `add` to SumModule
- Function `flatten` to module Flatten
- Set all modules to normal mode (instead of inplace, which does not work with backward hooks)
- Remove multiple usages of same module: This happens in ResNets most of the time with activation functions like ReLU. Because BackPACK cannot handle this repeated usage it creates a copy of the module.

As a result the computation graph consists exclusively of modules, and is therefore BackPACK-supported.

3.2 RNNs: Using PyTorch's Modules

Architecture Yu et al. (2019) review recent developments in RNN architectures. RNNs are commonly used for sequential data, such as video, audio, and text. According to the authors, the research on recurrent neural networks is now focused on LSTMs, introduced by Hochreiter and Schmidhuber (1997). This is a consequence of the success LSTMs have had in areas such as speech recognition. For example, He and Droppo (2016) propose a generalized LSTM layer that includes ideas from DNNs and achieve better results than both conventional DNNs and simple LSTMs. Another example is Hsu et al. (2016) who introduce skip connections between LSTM layers which is a similar approach like in residual networks. However, a recent comparison between RNN architectures by Shewalkar (2019) shows that simpler RNNs like GRUs can achieve similar accuracy with less training effort.

Challenges The recurrent structure poses several problems to be solved in BackPACK:

- Supporting modules: The first challenge is that the RNN modules from PyTorch don't have BackPACK support.

- ReduceTuple: The second challenge lies in the interface of these modules. Their forward pass returns a tuple, consisting of the hidden states from the last layer, as well as all hidden states of the last time step. This is different from most other modules that return a single tensor. Most applications only use one of the returned tensors, namely the hidden states of the last layer. However, the selection within the tuple is an operation that must be supported in BackPACK.

- Loop over module: The third challenge is to support loops over the same module. Currently, looping over a single module has two issues. During the forward loop, the saved input and outputs are overwritten. During the backward pass, the computed quantities from the previous iteration are overwritten.

Supporting modules Supporting PyTorch's RNN modules involves the computations laid out in section 2.2.1 and 2.2.2 and these are implemented. However, the resulting formulas involve the hidden states. Unfortunately, for computing the derivatives, the quantities that BackPACK is able to retrieve from the forward pass are insufficient. Therefore, BackPACK must perform an additional forward pass before the backward pass. However, considering that the most expensive operation is the actual backward pass it is still fine, especially compared to computing e.g. individual gradients with a for-loop.

ReduceTuple Selecting from the output tuple one entry is a process most users are not aware of. It is implicitly done with the built-in `getitem` function. The new custom module ReduceTuple does just the same as `getitem`, but has BackPACK support. The user must use this equivalent custom module instead of the built-in `getitem` function. In this way, BackPACK supports tuple selection.

Loop over module Looping over the same module is not possible in the scope of this thesis. The aforementioned obstacles are impossible to overcome in BackPACK's current structure.

The difficulties around saving the computed quantities can be solved for some extensions. For example, for computing the individual gradients, a routine that sums up the existing and new quantities leads to the correct behaviour. However, for other extensions, like variance, it is impossible to determine the new variance. The reason is that the variance we would like to calculate is $\mathrm{Var}_n(\{\sum_t \nabla_{\boldsymbol{\theta}_t^{(i)}} \ell_n\}_{n=1}^N)$ (where $\boldsymbol{\theta}_t^{(i)}$ denotes the parameter for each time step although they don't depend on time steps). Contrasting to this, in each time step, we only have access to $\mathrm{Var}_n(\{\nabla_{\boldsymbol{\theta}_t^{(i)}} \ell_n\}_{n=1}^N)$. One could think that summing up the variances from each time solves the issue, but the following equation for variance holds only for uncorrelated random variables.

$$\mathrm{Var}_n(\{\sum_t \nabla_{\boldsymbol{\theta}_t^{(i)}} \ell_n\}_{n=1}^N) \neq \sum_t \mathrm{Var}_n(\{\nabla_{\boldsymbol{\theta}_t^{(i)}} \ell_n\}_{n=1}^N)$$

But certainly, the variances of gradients from different time-steps are strongly correlated.

The difficulties around saving the quantities from the forward pass could also be solved. However, this would require a change how these quantities are saved. Additionally, it would require the user to determine the number of iterations. This is a major inconvenience and would lead to users shying away from this feature.

Converter function As mentioned before, creating a BackPACK-supported RNN involves using BackPACK's custom functions and limiting oneself to single-layer RNNs. This effort cannot be expected from all users and would limit the spread and usage of BackPACK. Similarly to ResNets, the converter function makes it possible to support as many architectures as possible. The important conversions are:

- ReduceTuple: The converter changes how the output is reduced. It replaces `getitem` with BackPACK's custom module ReduceTuple, which does just the same in the form of a module.
- Multi-layer RNNs: They are split into multiple RNNs with a single layer each. In PyTorch, with multiple layers, it is also possible to implicitly define intermediate dropout layers by giving a positive dropout probability. This is also handled by the converter function by adding these intermediate dropout layers explicitly. For example a module `nn.LSTM(5, 10, num_layers=2, dropout=0.2)` would be converted to `nn.Sequential(nn.LSTM(5,10), nn.Dropout(0.2), nn.LSTM(10,10))`.
- Permute: Convert functions `permute` and `transpose` to the new custom module Permute.

Chapter 4

Achievements

4.1 Supported Architectures

BackPACK is based on modules. Therefore, it is essential which modules have BackPACK support. We have created two tables illustrating which modules are generally supported and what has changed between the `master` branch and the `development` branch (September 2021). These changes are not necessarily part of this thesis.

Modules with parameters Table 4.1 lists the modules with parameters and which extensions can be used with them. From these modules, the BatchNorm, RNN, LSTM, and Embedding module have been a focus of this thesis. The detailed changes can be looked up in section 2.1 and 2.2. As can be seen in the table there have been major improvements for these modules.

The extensions are sorted in groups. The first order extensions BatchGrad, BatchL2Grad, Variance, and SumGradSquared are listed first. These extensions can be run independently of whether the rest of the neural network is supported. The rest of the extensions are second order extensions and require the whole network to be supported by BackPACK, including the modules without parameters.

Modules without parameters The modules without parameters are listed in table 4.2. The first order extensions are left out because they don't require any computations by these modules. However, for second order extensions it is essential that every single module has a BackPACK extension. This is especially important for the two loss functions MSELoss and CrossEntropyLoss.

Module	BatchGrad	BatchL2Grad	Variance	SumGradSquared	BatchDiagGGNExact	BatchDiagGGNMC	DiagGGNExact	DiagGGNMC	DiagHessian	BatchDiagHessian	GGNMP	HMP	KFAC	KFLR	KFRA	PCHMP
Conv1d	🟩	🟩	🟩	🟩	🟦	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥
Conv2d	🟩	🟩	🟩	🟩	🟦	🟦	🟩	🟩	🟩	🟦	🟩	🟩	🟩	🟩	🟩	🟩
Conv3d	🟩	🟩	🟩	🟩	🟦	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥
BatchNorm1d	🟩	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟩	🟩	🟥	🟥	🟥	🟥
BatchNorm2d	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
BatchNorm3d	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
RNN	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
LSTM	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
Linear	🟩	🟩	🟩	🟩	🟦	🟦	🟩	🟩	🟩	🟦	🟩	🟩	🟩	🟩	🟩	🟩
Embedding	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥

Table 4.1: Modules with parameters and which BackPACK extension are supported. Already supported modules in the `master` branch are highlighted in green, not supported modules are highlighted in red, and newly supported modules are highlighted in blue.

4.1.1 RNNs

Supported Architectures BackPACK supports several RNNs, including the Char-RNN from DeepOBS (Schneider et al., 2019) that is used for learning the Tolstoi dataset. At its core, the Char-RNN consists of a two-layer LSTM. An implementation in tensorflow can be found in the DeepOBS framework. During the master thesis, we wrote the required files for PyTorch and they will soon be added to DeepOBS. To reach this result, we have implemented core RNN modules, additional modules, and custom modules.

Core RNN modules From the wide range of RNN models, a lot of them use PyTorch's RNN modules (RNN, GRU, LSTM, RNNCell, GRUCell, and LSTMCell). As shown in table 4.1 the RNN, and LSTM module have support in the most common BackPACK extensions. The GRU module is complexity-wise in between RNN and LSTM and can be added if necessary. The cell versions could easily be supported, since they are just simpler versions of the other modules. However, to be useful, they must be included in a for-loop and this structure is incompatible with BackPACK.

Module	BatchDiagGGNExact	BatchDiagGGNMC	DiagGGNExact	DiagGGNMC	DiagHessian	BatchDiagHessian	GGNMP	HMP	KFAC	KFLR	KFRA	PCHMP
MaxPool1d	🟦	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥
MaxPool2d	🟦	🟦	🟩	🟩	🟩	🟦	🟩	🟩	🟩	🟩	🟩	🟩
MaxPool3d	🟦	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥
AvgPool1d	🟦	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥
AvgPool2d	🟦	🟦	🟩	🟩	🟩	🟦	🟩	🟩	🟩	🟩	🟩	🟩
AvgPool3d	🟦	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥
AdaptiveMaxPool1d	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
AdaptiveMaxPool2d	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
AdaptiveMaxPool3d	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
AdaptiveAvgPool1d	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
AdaptiveAvgPool2d	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
AdaptiveAvgPool3d	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
ELU	🟦	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥
LeakyReLU	🟦	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥
ReLU	🟦	🟦	🟩	🟩	🟩	🟦	🟩	🟩	🟩	🟩	🟩	🟩
SELU	🟦	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥
Tanh	🟦	🟦	🟩	🟩	🟩	🟦	🟩	🟩	🟩	🟩	🟩	🟩
Softmin	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
Softmax	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
Softmax2d	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
Identity	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
Dropout	🟦	🟦	🟩	🟩	🟩	🟦	🟩	🟩	🟩	🟩	🟩	🟩
MSELoss	🟦	🟦	🟩	🟩	🟩	🟦	🟩	🟩	🟩	🟩	🟩	🟩
CrossEntropyLoss	🟦	🟦	🟩	🟩	🟩	🟦	🟩	🟩	🟩	🟩	🟩	🟩
Flatten	🟦	🟦	🟩	🟩	🟩	🟦	🟩	🟩	🟩	🟩	🟩	🟩

Table 4.2: Modules without parameters and which BackPACK second order extension are supported. Already supported modules in the `master` branch are highlighted in green, not supported modules are highlighted in red, and newly supported modules are highlighted in blue.

Additional modules Embedding and multi-dimensional CrossEntropyLoss are required for RNNs. These are now supported in BackPACK

Custom modules Furthermore, for the functions `getitem`, `transpose`, and `permute` there is a BackPACK custom module equivalent. This is necessary to run second order extensions. For example the output of a RNN module that is a tuple is now reduced with the help of the ReduceTuple module instead of the `getitem`-function. The `transpose` and `permute` functions are needed to swap category and time axis before the loss.

4.1.2 ResNets

Supported Architectures All ResNets from torchvision are supported. These models include networks like resnet18 (He et al., 2016) and WideResNets (Zagoruyko and Komodakis, 2016). Especially the Wide ResNet 16-4 implemented in DeepOBS (Schneider et al., 2019) is supported. The components required to reach this coverage are the ResNet modules, handling branches, and introducing the converter function.

ResNet modules BatchNorm and AdaptiveAvgPool are modules that are used in ResNets but were previously incompatible with BackPACK. These modules are now supported to the extent required. BatchNorm works for all dimensions and in evaluation mode. AdaptiveAvgPool works for the required target sizes.

Branches The inplace summation `x+=submodel(x)` is at the core of residual networks. It means that the computation graph branches out and merges later with the summation. BackPACK can handle both the branching and the merging. The branching is handled implicitly without any user action required. The merging is handled with the SumModule instead of the built-in `add` function. This requires the user to use the SumModule.

Converter function The converter function is easy to use and takes care of all of these aspects.

4.2 Benchmarks

It is important to benchmark the new architectures in BackPACK, because we want to offer a solution that is applicable to realistic architectures and computes efficiently.

4.2.1 Setup

Airspeed velocity The benchmarks are produced with airspeed velocity (asv). The tool is originally intended for benchmarking a project over its lifetime. In this way, developers are able to notice drops in performance and can identify the commits that are to blame for it. This master thesis looks exclusively at the current state and benchmarks architectures that are representative for ResNets and RNNs.

Measurements The benchmark measures the time that one forward and one backward pass takes. Before the test, the architecture is loaded together with dummy data. The architecture is extended and converted with BackPACK's `extend()` function. After this preparation, the forward and the backward pass is performed and its time is measured. Finally, everything can be deleted and the next test is run. Luckily, there already existed a framework written for benchmarking BackPACK with asv. This framework has been adjusted to the new architectures and BackPACK's extended functionality.

Architectures

- Wide ResNet 16-4 from Schneider et al. (2019). This network was introduced by Zagoruyko and Komodakis (2016).
- Character RNN with two LSTM layers from Schneider et al. (2019). This architecture is used to learn the Tolstoi data set, which is simply the book War and Peace by Leo Tolstoi. The task is to predict the next character in the text from the previous characters.

Procedure

- Checkout schaefertim/backpack-benchmark on github (private but we grant access on request) and setup the project.
  In the future, this repository will probably be merged into
  f-dangel/backpack-benchmark on github
- Prepare machine: close all applications actively using CPU.
- run new benchmarks with
  ```
  asv run --launch-method=spawn --bench=time.*(tolstoi_char_rnn|svhn_wrn)
  ```

Machine The machine used for the benchmarks has the following specifications:

- RAM: 2 x 16 GiB: DIMM DDR4 Synchronous 2400 MHz (0,4 ns)

- L1 cache 384KiB, L2 cache 1536KiB, L3 cache 12MiB
- CPU: Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz
- GPU: Nvidia TU102 [GeForce RTX 2080 Ti Rev. A] (11 GB memory, 1350 MHz)
- motherboard: Z390 I AORUS PRO WIFI-CF

Baselines The benchmarks involve several baselines:

- torch: PyTorch without using BackPACK
- torch+extend: using BackPACK but without any extensions
- torch-BatchGrad-forLoop: individual gradients with PyTorch with a for-loop. The problems have batch size 128. Therefore, in the worst case, this is 128 times slower than torch.

4.2.2 Tolstoi Char RNN

Figure 4.1 shows the benchmark of the Tolstoi Character RNN problem from DeepOBS (Schneider et al., 2019). The baseline is chosen as only running torch. The other baseline torch+extend is quite similar and differs only because of the small absolute values. For this network, most extensions produce a significant overhead. The BatchGrad extension is approximately 20 times slower than the baseline. However, this is still faster than the alternative of computing individual gradients with a for-loop. For the other first order extensions (BatchL2Grad, SumGradSquared, Variance) a similar argument holds. The increased time effort of Variance compare to the BatchGrad extension is surprising because these two extension should have a very similar overhead. This implies that there might still be issues with the efficiency of the code. The DiagGGNExact extension takes about 120 times more time than the baseline. This is partly because its computation time scales with the number of free axis in the loss. In this particular architecture, the loss is multi-dimensional cross entropy loss with batch and time axis as the free axes. Therefore, this problem scales very badly for this extension. Nonetheless, the DiagGGN can now be computed in feasible time.

Possible improvements These runtimes are sufficient but not as convincing as we would have hoped. There are some possibilities for improvements:

- Currently, in LSTMs, BackPACK computes a full forward and backward pass for each Jacobian wrt the parameters and the input.

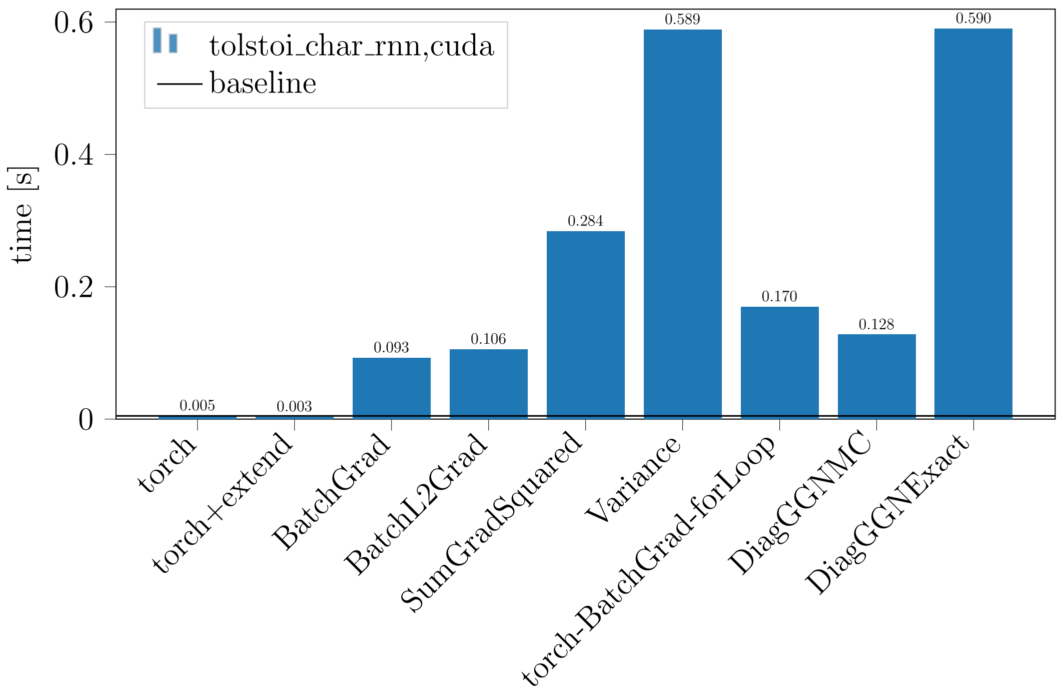


Figure 4.1: Benchmark Tolstoi Character RNN

These five runs can be reduced to a single run up to the product $\widetilde{IFGO}$ as was seen in section 2.2.2. However, this requires significant changes to BackPACK's core but could reduce the overhead by a factor of five.

- In the default LSTM implementation the time axis is the first axis, but most RNNs use the option `batch_first=True`. We don't expect any speed-up from changing this, but it might be the source of inefficiencies.

- There could also be other reasons that need to be tracked down with careful time measurements. The increased time consumption by the Variance extension is an indicator that other parts might be inefficient.

4.2.3 Wide ResNet 16-4

Figure 4.2 shows the benchmark of Wide ResNet 16-4 (Zagoruyko and Komodakis, 2016) from DeepOBS (Schneider et al., 2019).

Compared to the baseline, the BatchGrad extension takes approximately seven times longer. This is fast compared to the twelve times longer the alternative computation with the for-loop takes. The other first order extensions take approximately the same time as the BatchGrad extension. Only the DiagGGNExact extension takes significantly more time (30 times

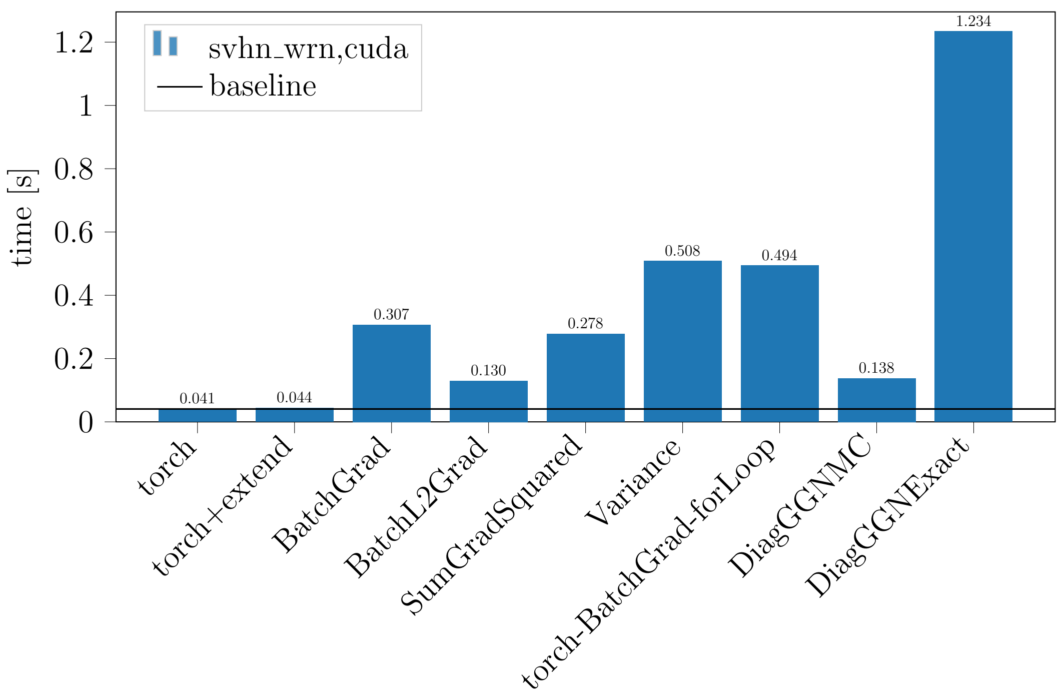


Figure 4.2: Benchmark ResNet

longer than baseline). This puts the DiagGGN into a feasible range and makes it useful for applications.

4.3 How to Use BackPACK

Figure 4.3 shows a code example of BackPACK with the converter. In the example, ResNet-18 (He et al., 2016) is loaded alongside some random input data similar to the imagenet data (Krizhevsky et al., 2009). The example consists of one forward and one backward pass. In the backward pass two additional quantities are computed in addition to the standard gradient:

- Variance (first order quantity)
- DiagGGN (second order quantity)

For the Variance extension, the standard BackPACK code is sufficient because it is a first order extension. For the DiagGGNExact extension, we must additionally use the converter because it is a second order extension.

Result The new required code for RNNs and ResNets is only `use_converter=True`. This is a minimal addition and grants access to a

```
X, y = torch.rand(32, 3, 224, 224, device="cuda"),
torch.randint(10, (32,), device="cuda")
model = torchvision.models.resnet18(num_classes=10).to("cuda")
model = extend(model.eval(), use_converter=True)
lossfunc = extend(CrossEntropyLoss())
loss = lossfunc(model(X), y)

with backpack(Variance(), DiagGGNExact()):
     loss.backward()

for param in model.parameters():
     print(param.grad)
     print(param.variance)
     print(param.diag_ggn_exact)
```

Figure 4.3: Code example for usage of BackPACK with the converter. Standard BackPACK code is highlighted in blue, converter is highlighted in red.

wide range of new networks. Current users of BackPACK are easily able to adapt this approach. Additionally, new users might be attracted by the new architectures available.

Chapter 5

Discussion and Outlook

5.1 BackPACK's role and Improvements

BackPACK's goal PyTorch computes exclusively the batch gradient. BackPACK wants to be a tool for researchers to try out and realize alternative approaches. Some of these approaches are laid out further below.

Strengths and weaknesses BackPACK offers convenient yet efficient access to additional quantities. This is achieved in the very few lines of code that must be added to use BackPACK and the converter. That BackPACK is very efficient in general and superior to naive PyTorch workarounds like the BatchGrad-forLoop has been shown by Dangel et al. (2019) and for the new architectures in section 4.2. The weakness of BackPACK is its small project size compared to PyTorch. This implies that BackPACK does not cover the full range of architectures and also that there is sometimes room for improvement in efficiency. Due to its small project size, BackPACK must take the flexible approach of fixing these holes on request.

Improvements in master thesis There have been major improvements to BackPACK during the master thesis. The main goal was to extend its coverage to ResNets and RNNs. This goal has been reached: ResNets have a wide range of support, especially common networks like those offered in torchvision. For RNNs, there are many different approaches. The core models that are based on LSTM modules are now supported. For example, the Character RNN that operates on the Tolstoi dataset (Schneider et al., 2019) is now supported. For both architecture classes this is a solid foundation. On this basis, many applications like possible training algorithms can prove their usefulness. If they prove to be useful on these common architectures, BackPACK can also be improved. This improvement can be either to cover more architectures or it can be to further improve its efficiency.

Project future The BackPACK project is very much alive and set up to be around in the future. It is used by the community (300 stars, 30 forks and multiple issues, such as questions and feature requests). Furthermore, BackPACK previously was in danger of not working with new PyTorch releases. This was because the core function `register_backward_hook` is deprecated and going to be deleted in favor of the new function `register_full_backward_hook`. This change of functions required several changes to BackPACK's core and secured the future of BackPACK. It is now expected to work with the next PyTorch releases.

5.2 Possible Applications

Here, we lay out some of the possible use cases for BackPACK. With BackPACK, these ideas are easier and faster to realize with potentially better results because of BackPACK's efficiency.

Training algorithms A major application are training algorithms. So far, the most successful training algorithms are purely based on the gradient. There are other ideas like using the natural gradient (Amari, 1998). The natural gradient is the gradient in the function space instead of the parameter space. This requires computation of the Fisher-matrix or approximations of it. Khan and Rue (2021) argue that this kind of optimization is the only efficient approach and currently popular algorithms are relaxations of this method. Dangel et al. (2019) show that these kind of algorithms can be implemented in BackPACK and the results are competitive with standard algorithms.

With BackPACK other training approaches are also possible. For example, hyperparameters can be implicitly determined from the additional quantities, yielding hyperparameter-free optimization. There are plenty of possibilities in this direction that can be tried out with the help of BackPACK.

Training observation Cockpit (Schneider et al., 2021) is a project that uses BackPACK. It offers many instruments to observe and visualize quantities during training. Therefore, it can ”improve troubleshooting the training process, reveal new insights, and help develop novel methods and heuristics”. This project will directly benefit from the implementation of RNNs and ResNets.

Robustness Other approaches try to improve robustness with additional quantities. For example Yao et al. (2018) use the largest eigenvalue of the Hessian matrix as a proxy of how sensitive the network is to adversarial attacks.

Minimizing this eigenvalue leads to a robust network. BackPACK can help this attempt by having efficient access to the Hessian or approximations of it.

Uncertainty measurement Laplace (Daxberger et al., 2021) can compute uncertainty measurements that are computationally cheap and competitive with other approaches. It uses BackPACK to compute different curvature approximations. This project also benefits from the increased range of architectures.

Future It is very uncertain which of these applications will emerge as successfully leveraging BackPACK. In this process, BackPACK will support researchers to try out new ideas and competitively compare them to standard approaches. The project will be shaped after the needs of the users and additional functionality will be added on request. Supporting ResNets and RNNs is a huge step for increasing the amount of applications and making BackPACK attractive to more researchers.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y. and Zheng, X. (2015), 'TensorFlow: Large-scale machine learning on heterogeneous systems'. Software available from tensorflow.org.

Amari, S.-I. (1998), 'Natural gradient works efficiently in learning', *Neural computation* **10**(2), 251–276.

Dangel, F., Kunstner, F. and Hennig, P. (2019), 'Backpack: Packing more into backprop', *CoRR* **abs/1912.10985**.

Daxberger, E., Kristiadi, A., Immer, A., Eschenhagen, R., Bauer, M. and Hennig, P. (2021), 'Laplace redux - effortless bayesian deep learning', *CoRR* **abs/2106.14806**.

Goodfellow, I., Bengio, Y. and Courville, A. (2016), *Deep learning*, MIT press.

He, K., Zhang, X., Ren, S. and Sun, J. (2016), Deep residual learning for image recognition, *in* 'Proceedings of the IEEE conference on computer vision and pattern recognition', pp. 770–778.

He, T. and Droppo, J. (2016), Exploiting lstm structure in deep neural networks for speech recognition, *in* '2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)', IEEE, pp. 5445–5449.

Hochreiter, S. and Schmidhuber, J. (1997), 'Long short-term memory', *Neural Computation* **9**(8), 1735–1780.

Hsu, W.-N., Zhang, Y., Lee, A., Glass, J. et al. (2016), 'Exploiting depth and highway connections in convolutional recurrent deep neural networks for speech recognition', *Cell* **50**, 1.

Ioffe, S. and Szegedy, C. (2015), Batch normalization: Accelerating deep network training by reducing internal covariate shift, *in* 'International conference on machine learning', PMLR, pp. 448–456.

Khan, M. E. and Rue, H. (2021), 'The bayesian learning rule', *arXiv preprint arXiv:2107.04562* .

Kingma, D. P. and Ba, J. L. (2015), Adam: A method for stochastic gradient descent, *in* 'ICLR: International Conference on Learning Representations', pp. 1–15.

Krizhevsky, A., Hinton, G. et al. (2009), 'Learning multiple layers of features from tiny images'.

Nesterov, Y. E. (1983), 'A method for solving the convex programming problem with convergence rate $o(1/k^2)$', *Dokl. Akad. Nauk SSSR* **269**, 543–547.

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

Polyak, B. T. (1964), 'Some methods of speeding up the convergence of iteration methods', *Ussr computational mathematics and mathematical physics* **4**(5), 1–17.

Robbins, H. and Monro, S. (1951), 'A stochastic approximation method', *The annals of mathematical statistics* pp. 400–407.

Rumelhart, D. E., Hinton, G. E. and Williams, R. J. (1986), 'Learning representations by back-propagating errors', *nature* **323**(6088), 533–536.

Sak, H., Senior, A. W. and Beaufays, F. (2014), 'Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition', *CoRR* **abs/1402.1128**.

Schneider, F., Balles, L. and Hennig, P. (2019), DeepOBS: A deep learning optimizer benchmark suite, *in* 'International Conference on Learning Representations'.

Schneider, F., Dangel, F. and Hennig, P. (2021), 'Cockpit: A Practical Debugging Tool for Training Deep Neural Networks'.

Shewalkar, A. (2019), ‘Performance evaluation of deep neural networks applied to speech recognition: Rnn, lstm and gru’, *Journal of Artificial Intelligence and Soft Computing Research* **9**(4), 235–245.

Tai, Y., Yang, J. and Liu, X. (2017), Image super-resolution via deep recursive residual network, *in* ‘Proceedings of the IEEE conference on computer vision and pattern recognition’, pp. 3147–3155.

Yao, Z., Gholami, A., Lei, Q., Keutzer, K. and Mahoney, M. W. (2018), ‘Hessian-based analysis of large batch training and robustness to adversaries’, *arXiv preprint arXiv:1802.08241* .

Yu, Y., Si, X., Hu, C. and Zhang, J. (2019), ‘A review of recurrent neural networks: Lstm cells and network architectures’, *Neural computation* **31**(7), 1235–1270.

Zagoruyko, S. and Komodakis, N. (2016), ‘Wide residual networks’, *arXiv preprint arXiv:1605.07146* .