Zenkai - Framework for Exploring Beyond Backpropagation

Greg Short
The Institute of Language and Speech Science
Waseda University
Tokyo, Japan
g.short@kurenai.waseda.jp

November 17, 2023

ABSTRACT

Zenkai is an open-source framework designed to give researchers more control and flexibility over building and training deep learning machines. It does this by dividing the deep learning machine into layers of semi-autonomous learning machines with their own target and learning algorithm. This is to allow researchers greater exploration such as the use of non-differentiable layers or learning algorithms beyond those based on error backpropagation.

Backpropagation Rumelhart et al. [1986] has powered deep learning to become one of the most exciting fields of the 21st century. As a result, a large number of software tools have been developed to support efficient implementation and training of neural networks through the use of backpropagation. While these have been critical to the success of deep learning, building frameworks around backpropagation can make it challenging to implement solutions that do not adhere to it. Zenkai aims to make it easier to get around these limitations and help researchers more easily explore new frontiers in deep learning that do not strictly adhere to the backpropagation framework.

Keywords deep learning · neural networks · backpropagation · target propagation · open source framework

1 Introduction

Deep learning research has boomed over the past 15 years making neural networks into one of the hottest topics in academia and industry. One huge contributor to this proliferation has been open source software tools like Torch7 (Collobert et al. [2011]), Theano (Theano Development Team [2016]), Tensorflow (Abadi et al. [2015]), Chainer (Tokui et al. [2019]), and PyTorch (Paszke et al. [2019]). This has made deep learning accessible not only machine learning researchers, but also to researchers in a wide-variety of other fields, to data scientists, to software engineers, and to hobbyists. They are mostly designed to be easy-to-use with an intuitive API that hides the details of automatic differentiation and learning and are also designed to be highly performant. They have also spawned ecosystems of second and third generation libraries and frameworks that have extended the functionality of the core framework like BoTorch (Balandat et al. [2020]), extending PyTorch to Bayesian optimization, or Pyro (Bingham et al. [2018]), a probabilistic programming framework built with PyTorch, or by improving the quality of life and usability like with Keras (Chollet et al. [2015]) or Lightning (Falcon and The PyTorch Lightning team [2019]).

The core tools for deep learning are primarily designed to learn through backpropropagation, the algorithm behind much of the advancements in deep learning Rumelhart et al. [1986]. This reliance on backpropagation has been a strength because it has allowed them to highly optimize the performance of the operation and also the usability by abstracting away the details. However, it can also be weakness because it makes it harder to explore different mechanics and learning algorithms for an individual layer, which increases the difficulty of researching topics that do not conform to the backpropagation framework. While in some cases, learning algorithms can still can still be found to cast the problem to the backpropagation framework like with straight-through-estimation Hinton [2012], Bengio [2013] or like with differentiable decision trees Silva et al. [2020], limits are still exist on what can be learned and how it learns.

Zenkai is an open source tool built on PyTorch Paszke et al. [2019], which has been developed to overcome these limitations in order to explore other architectures and learning mechanics. This report on Zenkai will be structured as follows. First, the core concepts of Zenkai and the primary design principles of Zenkai will be discussed. Then, the features Zenkai provides will be discussed in more detail. Following that, several experiments will be presented to demonstrate some of the possibilities for what can be done with Zenkai. Lastly, the conclusion and future work will be given. The repository is accessible on Github [1] with documentation [2] and can be installed with pip [3].

2 Concept

Zenkai is designed to stimulate exploration in deep learning research by dividing a learning machine into layers of semi-autonomous learning machines. It views a learning machine with N hidden layers as being composed of $N + 1$ layers of learning machines depth-wise. And if necessary each of these layers can be potentially be divided into M learning machines width-wise where M is the number of functions generating the outputs for a layer. By dividing it into layers which each have their own `step()` and `step_x()` methods, the researcher can more freely decide on the learning algorithm for each layer or subset of layers.

This concept is inspired by target propagation Bengio [2014], Le Cun et al. [1988], Lee et al. [2014], Meulemans et al. [2020] in which targets are propagated backward through the network rather than error gradients and each layer of a neural network has its own target. In target propagation, each target is obtained by using an inverse operation or approximation to the inverse such as the pseudoinverse or a reconstruction operation like in an autoencoder. To illustrate how this works in Zenkai, Code 2 is shown with a dummy class that inherits from the primary class of Zenkai, the LearningMachine.

```python
class DummyLearningMachine(LearningMachine):

    def forward(self, x: IO, state: State, release: bool=True) -> IO:
        # use freshen to detach and also set retain_grad to True if not
        # freshened yet
        x.freshen()
        return x

    def assess_y(self, x: IO, t: IO, state: State) -> Assessment:
        return self.loss.assess(x, t, state)

    def accumulate(self, x: IO, t: IO, state: State) -> Assessment:
        # accumulate parameter updates. This step is not necessary

    def step(self, x: IO, t: IO, state: State):
        # update parameters

    def step_x(self, x: IO, t: IO, state: State) -> IO:
        # determine the target for the incoming layer
```

The method `forward()` is the standard forward method inherited from `nn.Module`. The method `assess_y()` is used to evaluate the output of the machine. The method `step()` updates the parameters of the learning machine. The method `step_x()` updates the input which can then be used as the target to the incoming machine. The `accumulate()` method allows the user to collect changes before being processed similar to `accGradParameters()`. The learning machines can then be stacked onto one another to form a deep learning machine.

Example Code 2 is meant to illustrates that. In this case, it is visible that `step()` method also serves as the way to propagate to lower layers. Each layer is updated and then the target for the preceding layer is computed with `step_x()`. Depending on the layer type, these can be reversed. The `accumulate()` method is not implemented here and not required. If it is implemented, however, the propagation down through layers will usually need to occur in that method, which removes the ability to alternate the order of `step_x()` and `step()`

```python
def KindofDeepLearner(LearningMachine):
    ...
    # go backward through the network and update the parameters
```

---

[1]Repository: https://www.github.com/short-greg/zenkai
[2]Documentation: http://zenkai.readthedocs.io
[3]Installation: `pip install zenkai`

```
4    def step(self, x: IO, t: IO, state: State):
5        my_state = state.mine(self, x)
6        # Here step is done before step_x. That means the parameter update is
             executed first
7        self.layer3.step(my_state.layer2, t)
8        # It may be desirable to do step_x first,
9        # like with backpropagation
10       t = self.layer3.step_x(my_state.layer2, t)
11       # Zenkai also offers convenience methods to reduce the
12       # lines of code like backward()
13       self.layer2.step(my_state.layer1, t)
14       t = my_state.t1 = self.layer2.step_x(my_state.layer1, t)
15       self.layer1.step(x, t)
16
17    # step() must be executed prior to step_x()
18    @step_dep('t1')
19    def step_x(self, x: IO, t: IO, state: State) -> IO:
20
21        # a dependency is set on step() and it is set so that step() will not be
22        # executed automatically. Thus an error will be raised if it hasn't
23        return self.layer1.step_x(x, state[self, x, 't1'])
```

Because each layer has its own target, pitfalls of using gradient descent such as not being able to train non-differentiable functions can be potentially avoided. Also, as can be seen this makes it easy to have the model parameters updated prior to obtaining the targets for the preceding layer. As mentioned above, target propagation can be used for this when using a trainable layer for reconstruction. The vanilla form of target propagation uses a shallow neural network to predict the targets of the prior layer by approximating the inverse.

$$t_h = N^{-1}(t_{h+1}) \tag{1}$$

However, gradient descent can also be easily cast to adhere to this framework as the new "target" of layer h_{i-1} can be calculated from the gradient that is backpropagated to the output of layer i. In fact, this approach is typically used to determine the targets for the output layer for target propagation. As can easily be shown, layer h_{i-1} backpropagates the sum of squared errors loss with new target value calculated by updating the output with the gradient.

$$t_h = y_h - \nabla N_{h+1}(y_h) \tag{2}$$

And since this can easily be converted to sum of squared errors loss through integration, standard error backpropagation is relatively easy to implement in Zenkai.

$$\text{SSE} = 0.5 \sum_{i=1}^{n} (y_{h_i} - t_{h_i})^2 \tag{3}$$

Using this approach would be inferior to just using PyTorch for a regular neural network as it would require more code and lose PyTorch's abstractions. However, this can be useful if some layers make use of gradient descent and others do not or the researcher wants to explore other local loss functions besides SSE for an intermediate layer. Another possibility in Zenkai is to loop over the minibatch and have multiple "local updates" for each "global update".

Zenkai offers more methods for obtaining the target for a layer. For instance, if the layer is non-differentiable, hill-climbing can be used. One possibility for hill climbing is to perturb the inputs and return the input that minimizes the error. In this paper, a variation of that idea is tested. A stochastic layer (using dropout) that precedes the layer is used to generate candidate targets, then the candidate that minimizes the error is chosen as the new target. It is also possible to connect the global loss more directly to the intermediate layer or simply to perturb the outputs of a layer and set the target to the candidate that produces the best results. While not all possible approaches are guaranteed to lead to minimizing the global cost function, Zenkai gives the researcher more freedom to explore different approaches.

Another benefit of this system is it is also relatively easy to calculate error reduction of the global cost function or a local loss function after each intermediate update of the parameters or the inputs. Global error reduction (GER) can be defined as the reduction in error of the global cost function after an update,

$$\text{GER} = L_g(N_{h_{pre}-o}(x_{h_{pre}}), t_g) - L_g(N_{h_{post}-o}(x_{h_{post}}), t_g) \tag{4}$$

where L_g represents the global cost, $N_{h_{pre}-o}$ represents the layers h to o of the network before the update, $x_{h_{pre}}$ represents the input of `layer` h, and *post* represents them after the update.

Local error reduction can be calculated as the reduction in error for a local cost function (i.e. the loss for an intermediate layer).

$$\text{LER} = L_h(N_{h_{pre}}(x_{h_{pre}}), t_h) - L_h(N_{h_{post}}(x_{h_{post}}), t_h) \tag{5}$$

where L_h represents the local loss, $N_{h_{pre}}$ represents `layer` h before the update, $x_{h_{pre}}$ represents the input of `layer` h, and *post* represents them after the update.

This can be helpful to identify the causes of unsuccessful learning. Error reductions tending to be negative might indicate instability which can lead to poor training. Local error reductions being quite small may indicate that the target differences are too small or the learning is too slow.

Through these features, the researcher can experiment with using other techniques such as using decision trees in place of neurons. The researcher can choose to propagate directly from the output layer as in Direct Feedback Alignment. The researcher can also use population-based optimizers to update the layer parameters as has been used in neural network research such as Salimans et al. [2017]. There are a multitude of possibilities.

3 Design and Features

3.1 Design Principles

As exploring new machine architectures and learning algorithms is the primary objective of Zenkai, the primary design objectives for Zenkai are flexibility and ease-of-use. Flexibility is important to be able to test a wide variety of solutions. Ease-of-use is important to keep the difficulty of exploration down. However, increasing flexibility tends to increase the difficulty of usage, because to add flexibility it is often necessary to increase the level of abstractions, but to increase ease-of-use, one must often reduce the level of abstraction. Zenkai mostly follows modular design principles to achieve this aiming for high cohesion and loose coupling. As this requires a balance, it allows the researcher to decide whether to emphasize coupling or cohesion. Also, in order to increase ease-of-use, the systems are implemented to aim for familiarity and consistency. For instance, many of the classes inherit from PyTorch's classes. Also, the learning flow from Torch7 was also mimicked typically with a four step process, `forward()`, `accumulate()`, `step_x()`, `step()`. `accumulate()` can be removed however, and if removed, the order of `step_x()` and `step()` can be reversed.

3.2 Features

Zenkai is broken down into six subpackages presently: kaku for the core features and interfaces, kikai for implementations of different types of learning Machines, tansaku for population-based metaheuristics, Mod for utilities that inherit from `nn.Module` and Utils for functions used by the framework.

3.2.1 Kaku (i.e. Core)

Kaku contains the features for defining a LearningMachine (2) and all of the components that go into one for updating the parameters, controlling the inputs and outputs, obtaining the targets, assessment, and so on. LearningMachine is the most important class in Zenkai as it defines an interface for implementing learning machines that can be connected to one another.

The core subpackage is imported by simply importing Zenkai 'import zenkai'. In addition, there are functionalities in kaku for callbacks for the learning machine, and definitions of criterions that can be used to calculate the loss or value.

Components

- **IO:** Wraps the tensors used for inputs, outputs and targets with a tuple.
- **StepTheta:** Used to update the parameters of the machine
- **StepX:** Used to update the x values to determine the targets of the preceding layer.
- **Assessment:** Contains an evaluation plus whether a cost or utility.

- **State:** Stores the state for the current learning step.

- **LearningMachine:** Learner/nn.Module that inherits from StepTheta and StepX.

- **Individual:** A class for defining one element of a population. Primarily used for metaheuristics.

- **Population:** A class for defining a group of individuals. Primarily used for metaheuristics.

3.2.2 Kikai (Machines)

Kikai contains the class definitions for a variety of LearningMachines, StepThetas or StepXs meant to be the building blocks of larger machines. There are classes for wrapping scikit-learn estimators, for Feedback and Direct Feedback Alignment Lillicrap et al. [2016], Nøkland [2016], Liao et al. [2015], Moskovitz et al. [2019], Refinetti et al. [2021], for ensembles, for reversible functions, for target propagation, and so on. It also contains classes like GraphLearner which is there to makes it easier to implement compositions of learners.

3.2.3 Tansaku (Search)

Tansaku is a collection of modules that allow one to build and execute population-based optimizers using PyTorch primarily to provide more techniques for updating the parameters or obtaining the targets for a LearningMachine. With it, the researcher can flexibly implement different types of such as hill climbing algorithms, genetic algorithms, particle swarm optimization algorithms, and so on. The core classes for it are Individual and Population as well as classes for to modifying an individual or population.

In general, a series of operations like the one in Code 3.2.3 will be used to implement the optimization algorithm. Here you can see a population is generated, it is modified and assessed and finally reduced to an individual.

```python
def climb_hill(self, individual: Individual) -> Individual:
    """Do randomized perturbations on the individual to
    get a population then get the best individual in the population """

    # create a population from the individual
    population = individual.populate(k=8)
    population = self.perturb(population)
    # evaluate each individual in the population
    population = self.assessor(population)
    # reduce the population to get the hopefully improved individual
    return self.reducer(population)
```

3.2.4 Utils and Mod

Zenkai also provides a variety of utilities and modules that are used by the rest of Zenkai. These include reversible modules for use by kikai's ReversibleMachine, functions for the setting and getting of model parameters and gradients, and for modules that allow for creating ensemble models, among more.

4 Example Experiments

In this section, I demonstrate several cases where I've used Zenkai to implement a variety of networks, focusing on ones that make use of techniques other than or in addition to backpropagation. These demonstrations are meant to give a preview of some of the possibilities for learning machines to implement with Zenkai. I have not used any systematic approaches to optimize the hyperparameters so the results in most cases probably do not approximate an upperbound on what is achievable. In some cases, like Feedback Alignment, better results have been demonstrated. Code is given for some of them to demonstrate how the method is implemented with Zenkai, but not all as including code for all of them would lengthen the document considerably. The details of the algorithms proposed in other research are not discussed as that is beyond the scope of this article.

The results are summarized in Table 4.7. For each experiment, the network is described, followed by possible use cases and the results. Finally, the network definition is given along with any other necessary details such as algorithms. The MNIST dataset Deng [2012] is used for all tests for simplicity and for comparison, even though it may not be ideal in some cases.

4.1 Baseline

First, a neural network using backpropagation created with the Zenkai framework is tested with two fully-connected "neural" layers. Since Zenkai requires targets to be passed backward, the targets are calculated by the formula in Equation 2 and the sum of squared error loss is computed and multiplied by 0.5 to backpropagate the gradients to the preceding layer. Details of the architecture and training are given in A.1.

4.2 Decision-Linear

Decision-Linear demonstrates the potential usage of decision trees as using decision trees could be beneficial because of their interpretability. This shows the use of a machine in which the first layer is composed of an operation other than matrix multiplication. The second layer in the network is a standard linear layer. Details of the architecture and training are given in A.2.

Scikit-learn Pedregosa et al. [2011] was used for the decision trees. Because `partial_fit()` is not available for scikit-learn's decision trees `fit()` is used instead. However, because in minibatch learning this will overfit to the minibatch, an ensemble of trees is used. Overall, the learning is quite slow since the components were not designed for online updating or optimally designed for outputting multiple values. An alternative is to use one decision tree that outputs multiple values but that has its own downsides like highly correlated outputs. Comparable results to the baseline are achieved, however.

The network uses an ensemble of estimators for the decision tree regressor layer. To form the ensemble for that layer, the nine most recent estimators are used. When fitting for a time step, the oldest estimator is dropped if the number of estimators has reached nine and a new layer is added. The new estimator is then fit. The minibatch size was increased as training progressed until the minibatch size reached the size of the batch.

Algorithm 1 Temporally Dependent Ensemble Updating

1: **if** ensemble.full() **then**
2:     ensemble.remove_index(0)
3: **end if**
4: new_estimator = copy(base_estimator)
5: new_estimator.fit(x, t)
6: ensemble.append(new_estimator)

In addition, gradient descent is repeated 40 times on the inputs of the output layer to get the targets for the first layer. Without this, the updates tended to be quite small which resulted in excessively slow training. While this implementation is slow, it is still quite preliminary using components that are not well-designed for this application.

4.3 Target Propagation: Regularized Least Squares

Here a neural network is trained that uses target propagation with regularized least squares to determine the target of the previous layer similar to Le Cun et al. [1988] or Roulet and Harchaoui [2021]). A potential benefit of using Least Squares is that it results in larger updates than gradient descent. In this example, it is only used for propagating the target, but it can be used for each layer in the network as well. Details are given in A.3.

To get the target of linear layer, ridge regression is used to find the change in x that minimizes the squared error with Scipy's linear algebra solver method Virtanen et al. [2020]. While that is used for the linear operation, gradient descent is used to calculate the targets for activations.

The results do not come close to the baseline, but a lot can be done to improve upon the results such as reducing the "learning rate" over time for target propagation. Also, it is possible to use least squares for parameter updates and Zenkai provides tools to filter the parameters for that.

4.4 Target Propagation: Reconstruction Layer

Another approach to target propagation is to make each layer be an autoencoder and to learn an approximation to the inverse (i.e. a reverse operation). The approximation is then used to calculate the target of the incoming layer. This is a more flexible formulation as it can potentially be used to train non-differentiable operations. In this network, I use error backpropagation for the output layer and target propagation for the remaining layers. Also, training is alternated between training the reverse model and training the forward model. Details are given in Section A.4. The implementation is loosely based on Bengio [2014].

In A.4, the basic code for implementing a layer using target propagation is shown, though this code removes some of the intricacies like training only the forward or reverse layer for simplification. The results were not impressive, however. The learning was a little unstable and the mean-absolute deviation between the inverted y and t for a target propagation layer tend not to shrink. With more sophisticated approaches such as Lee et al. [2015], better results should be achievable. Also, learning is alternated between training the reverse model for one epoch and the forward model for the following epoch.

4.5   Feedback Alignment

Feedback alignment Lillicrap et al. [2016] is a biologically plausible training algorithm for neural networks that makes use of a randomly generated weight matrix used for backpropagation, which stays fixed. This example shows Zenkai's flexibility in being able to be used when backpropagation does not follow the standard form. Details of the architecture and training are given in A.5.

Zenkai makes it more straightforward to implement because the `step_x()` method is user-defined. While there are ways to implement this with only PyTorch, it will possibly result in implementing an AutoGrad functor and defining the backward() method to use a random matrix which is passed in.

Feedback alignment updates the weights with the following formula

$$\delta W_i = -((\mathbf{B}_{i+1}\delta z_{i+1}) * \phi'(z_i))y_{i-1}^T \qquad (6)$$

where B is a randomly generated, constant, δz_{i+1} is the gradient from the outgoing layer, $\phi'(z_i))$ is the activation and y_{i-1}^T are the outputs of the preceding layer.

These results obtained are not on par when compared to the baseline or in terms of other works but this can be expected with little hyperparameter tuning. Also, Zenkai offers functionality to use other operations besides Torch's `nn.Linear` in Feedback Alignment.

4.6   Direct Feedback Alignment (DFA)

Direct feedback alignment Nøkland [2016] is a variation of feedback alignment that has a direct connection from each layer to the output layer on the backward pass. This experiment using it illustrates another use of learning that can be challenging to implement with standard frameworks due to the non-standard backward pass. Details of the architecture and training are given in A.5.

For updates, direct feedback makes use of the matrix **B** having the shape [N, O] where N is the number of outputs of the layer and O is the number outputs in the network. Otherwise the formula for updating is similar to Feedback Alignment. The implementation is straightforward with Zenkai because the `step()` method for the network simply calls the step method for each layer without using `step_x()`. As with regular feedback alignment, the type of underlying operation can be decided on by the researcher. In this experiment, though, a standard Linear layer was used.

4.7   Neural Decision

This network is composed of a convolutional network followed by a non-differentiable decision tree classifier for the outermost layer. This demonstrates the possibilities for using a non-differentiable layer in a position that requires propagation to lower layers. Details of the architecture and training are given in A.7.

To get around the lack of differentiability, the targets for the convolutional network are computed with a hill climbing algorithm. Candidate targets are generated by the convolutional network by passing each sample through it K times. Since the first layer uses dropout, it is stochastic so K different samples are generated for each instance in the minibatch. The samples that produced the highest average classification rate in the ensemble for each instance for the decision tree layer are chosen to be the targets. The basic algorithm is defined below, though the implementation makes use of vectorized operations rather than loops.

This produces okay results, but the results did not reach the baseline despite using convolutional operations. Like with target propagation, this resulted in large mean absolute deviations between the outputs and the targets. To improve the results, it is possibly necessary to reduce the variance of the candidate targets that are produced by gradually reducing the dropout probability or some other means, but I have decided to leave that to future work.

7

Algorithm 2 Hill-climbing with Stochastic Incoming Layer

1: **repeat**
2:    Repeat the input i K times
3:    Compute the output of the incoming layer
4:    Reshape the output so that the population dimension is first and the sample dimension is second
5:    Choose the output that maximizes the average classification accuracy as the target
6: **until** until looped over all inputs

Table 1: Accuracy Results of Experiments

	Exp 1	**Exp 2**	**Exp 3**	**Exp 4**	**Exp 5**	**Exp 6**	**Exp 7**
Experiment	Baseline	Decision-Linear	Least Squares	Linear Target Prop	Feedback Alignment	Direct Feedback Alignment	Neural Decision
Accuracy	0.969	0.967	0.941	0.929	0.958	0.940	0.939

5   Conclusion

In this document, I have introduced Zenkai, a framework to stimulate deep learning research beyond simple backpropagation. It does this by making it easier and giving more freedom for the researcher to define when and how the hidden layers will be updated and also what the internal mechanics of the layer will be.. This framework aims to provide researchers with more flexibility in architecture and training design for machines with hidden layers.

I have described the concepts and design principles, which are to provide flexibility and ease-of-use. I have also described the features contained in Zenkai and its subpackages, kikai, tansaku. I then presented several illustrative examples through experiments meant to provide some ideas for what can be accomplished with it with some using non-differentiable operations. These examples showed a variety of possible architectures that can be difficult to implement with standard deep learning frameworks. And there are many more possibilities for what can be implemented with it.

For future work, I plan to create more demonstrations to show how Tansaku can be used effectively. I also intend to expand on the capabilities of the subpackage of `zenkai.tansaku` and continue to add more features to increase ease of use. I also intend to use Zenkai for work on Deep Neurofuzzy systems, a type of deep learning machine that is difficult to optimize with backpropagation when based on max and min operations in place of addition and multiplication.

References

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986. URL https://api.semanticscholar.org/CorpusID:205001834.

Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *NIPS 2011*, 2011. URL https://api.semanticscholar.org/CorpusID:14365368.

Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL http://arxiv.org/abs/1605.02688.

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, Shuji Suzuki, Kota Uenishi, Brian Vogel, and Hiroyuki Yamazaki Vincent. Chainer: A deep learning framework for accelerating the research cycle, 2019.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison,

Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

Maximilian Balandat, Brian Karrer, Daniel R. Jiang, Samuel Daulton, Benjamin Letham, Andrew Gordon Wilson, and Eytan Bakshy. BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization. In *Advances in Neural Information Processing Systems 33*, 2020. URL http://arxiv.org/abs/1910.06403.

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep universal probabilistic programming, 2018.

François Chollet et al. Keras. https://keras.io, 2015.

William Falcon and The PyTorch Lightning team. PyTorch Lightning, March 2019. URL https://github.com/Lightning-AI/lightning.

Geoffrey Hinton. Neural networks for machine learning. *Neural networks for machine learning*, 2012.

Yoshua Bengio. Estimating or propagating gradients through stochastic neurons. *CoRR*, abs/1305.2982, 2013. URL http://arxiv.org/abs/1305.2982.

Andrew Silva, Taylor Killian, Ivan Dario Jimenez Rodriguez, Sung-Hyun Son, and Matthew Gombolay. Optimization methods for interpretable differentiable decision trees in reinforcement learning, 2020.

Y. Bengio. How auto-encoders could provide credit assignment in deep networks via target propagation. 07 2014.

Yann Le Cun, Conrad Galland, and Geoffrey E Hinton. Gemini: Gradient estimation through matrix inversion after noise injection. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 1. Morgan-Kaufmann, 1988. URL https://proceedings.neurips.cc/paper_files/paper/1988/file/a0a080f42e6f13b3a2df133f073095dd-Paper.pdf.

Dong-Hyun Lee, Saizheng Zhang, Antoine Biard, and Y. Bengio. Target propagation. 12 2014.

Alexander Meulemans, Francesco S. Carzaniga, Johan A. K. Suykens, João Sacramento, and Benjamin F. Grewe. A theoretical framework for target propagation. *CoRR*, abs/2006.14331, 2020. URL https://arxiv.org/abs/2006.14331.

Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning, 2017.

Timothy P. Lillicrap, Daniel Cownden, Douglas Blair Tweed, and Colin J. Akerman. Random synaptic feedback weights support error backpropagation for deep learning. *Nature Communications*, 7, 2016. URL https://api.semanticscholar.org/CorpusID:10050777.

Arild Nøkland. Direct feedback alignment provides learning in deep neural networks, 2016.

Qianli Liao, Joel Z. Leibo, and Tomaso A. Poggio. How important is weight symmetry in backpropagation? *CoRR*, abs/1510.05067, 2015. URL http://arxiv.org/abs/1510.05067.

Theodore H. Moskovitz, Ashok Litwin-Kumar, and L. F. Abbott. Feedback alignment in deep convolutional networks, 2019.

Maria Refinetti, Stéphane d'Ascoli, Ruben Ohana, and Sebastian Goldt. Align, then memorise: the dynamics of learning with feedback alignment, 2021.

Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

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

Vincent Roulet and Zaid Harchaoui. Target propagation via regularized inversion, 2021.

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi:10.1038/s41592-019-0686-2.

Dong-Hyun Lee, Saizheng Zhang, Asja Fischer, and Yoshua Bengio. Difference target propagation, 2015.

A   Learning Machine Definitions

Below, the definitions for the machines used in the experiments are given.

A.1   Baseline

This architecture defines a standard fully connected neural network making use of Zenkai. This approach can be used to connect a neural network that uses gradient descent to some other type of machine.

Layer 1: Fully Connected Neural Layer

1. Architecture: 128 units, ReLU Activation, BatchNorm
2. Update: Gradient Descent, Adam Optim, 1e-3 LR, SSE Loss
3. X Update: N/A

Layer 2: Fully Connected Neural Layer

1. Architecture: 32 Units, ReLU Activation, BatchNorm
2. Update: Gradient Descent, Adam Optim, 1e-3 LR, SSE Loss
3. X Update: Gradient Descent (from Update)

Layer 3: Fully Connected Neural Layer

1. Architecture: 10 Units
2. Update: Gradient Descent, Optim=Adam, LR=1e-3, Cross Entropy Loss
3. X Update: Gradient Descent (from Update)

A.2   Decision-Linear Definition

This machine has two layers, a layer composed only of decision tree regressors and a standard linear layer. The decesion tree regressors were trained using scikit-learn's `DecisionTreeRegressor` along with its `MultiOutput` so for each iteration of training 32 decision trees are learned.

Layer 1: Decision Tree Regressor Layer

1. Architecture: Ensemble of 9 / 32 units Decision Trees -> Voting
2. Update: 11 Max Depth, scikit-learn defaults
3. X Update: N/A

Layer 2: Fully Connected neural Layer

1. Architecture: BatchNorm -> 32 Unit Linear -> ReLU Activation,
2. Update: Gradient Descent - Adam Optim, 1e-3 LR, Cross Entropy Loss, 128 batch size
3. X Update: Gradient Descent - 40 iterations

A.3   Target Propagation: Regularized Least Squares Definition

The first version of target propagation uses 4 linear layers. The target is propagated backward through the linear layer using the pseudoinverse. For the activations, however, gradient descent was used.

Layer 1: Fully Connected Neural Layer

1. Architecture: 128 units Linear -> BatchNorm -> ReLU activation
2. Update: Gradient Descent - Adam Optim, 1e-3 LR, SSE Loss
3. X Update: N/A

Layer 2: Fully Connected Neural Layer

1. Architecture: 128 units Linear -> BatchNorm -> ReLU activation
2. Update: Gradient Descent - Adam Optim, 1e-3 LR, SSE Loss
3. X Update: Regularized Least Squares

Layer 3: Fully Connected Neural Layer

1. Architecture: 128 units Linear -> BatchNorm -> ReLU activation
2. Update: Gradient Descent - Adam Optim, 1e-3 LR, SSE Loss
3. X Update: Regularized Least Squares

Layer 4: Fully Connected Neural Layer

1. Architecture: 10 units Linear
2. Update: Gradient Descent - Adam Optim, 1e-3 LR, Cross Entropy Loss
3. X Update: Regularized Least Squares

A.4 Semi-Target Prop

This variation of target propagation uses autoencoders for each target propagation layer. This machine has 4 linear layers, the outermost of which uses gradient descent for obtaining the incoming target.

Layer 1: Fully Connected Linear Layer

1. Architecture: 128 Units Linear -> ReLU
2. Update: Gradient Descent, Adam, 1e-3 lr
3. X Update: N/A

Layer 2: Fully Connected Linear Layer

1. Architecture: 128 Units Linear -> ReLU
2. Update: Gradient Descent, Adam, 1e-3 lr, SSE Loss
3. Reverse Architecture: 128 Units, BatchNorm, ReLU / 128 Units, BatchNorm, ReLU
4. Reverse Update: Adam, 1e-3 lr, MSE Loss
5. X Update: Reverse(t)

Layer 3: Fully Connected Linear Layer

1. Architecture: 128 Units Linear -> ReLU
2. Update: Adam, 1e-3 lr, SSE loss
3. Reverse Architecture: 128 Units, BatchNorm, ReLU / 128 Units, BatchNorm, ReLU
4. Reverse Update: Adam, 1e-3 lr, MSE Loss
5. X Update: Reverse(t)

Layer 4: Fully Connected Linear Layer

1. Architecture: 10 units Linear
2. Update: Gradient Descent, Adam, 1e-3 lr
3. X Update: Gradient descent

Here is an example of the code that can be used to implement a target propagation layerf or this example.

```
class LinearTargetPropLearner(zenkai.LearningMachine, TargetPropStepX):

    def __init__(
        self, in_features: int, out_features: int,
        lr: float=1e-3, reduction: str="mean", loss="cross_entropy",
```

```
 6            name: str="TargetProb"
 7        ):
 8            super().__init__()
 9            self.layer = nn.Sequential(
10                nn.Linear(in_features, out_features),
11                nn.BatchNorm1d(out_features),
12                nn.ReLU(),
13            )
14            reverse_layer = nn.Sequential(
15                nn.Linear(out_features, in_features),
16                nn.BatchNorm1d(in_features),
17                nn.ReLU(),
18                nn.Linear(in_features, in_features),
19                nn.BatchNorm1d(in_features),
20            )
21            self._target_prop_learner = GradLearner(
22                reverse_layer, ThLoss('mse', reduction="mean"), OptimFactory('adam', 1
                     e-3), "mean"
23
24            )
25            self._step_theta = zenkai.kikai.GradStepTheta(self, zenkai.itadaki.adam(lr
                 =lr), reduction)
26            self._loss = zenkai.ThLoss(loss, reduction=reduction, weight=0.5)
27
28        def step_target_prop(self, x: IO, t: IO, y: IO, state: State):
29            self._target_prop_learner.step(y, x, state)
30
31        def accumulate(self, x: IO, t: IO, state: State):
32            self._step_theta.accumulate(x, t, state)
33            x_prime = self._target_prop_learner(state[self, 'y'], state)
34            self._target_prop_learner.accumulate(state[self, 'y'], x.detach(), state)
35
36        def step_x(self, x: IO, t: IO, state: State) -> IO:
37            x_prime = self._target_prop_learner(t, state, release=True)
38            return x_prime
39
40        def assess_y(self, y: IO, t: IO, reduction_override: str = None) ->
               AssessmentDict:
41            return self._loss.assess_dict(y, t, reduction_override)
42
43        def forward(self, x: IO, state: State, release: bool = True) -> IO:
44            x.freshen()
45            y = state[self, 'y'] = self.layers(x)
46            return y.out(release)
47
48        def step(self, x: IO, t: IO, state: State):
49
50            self._step_theta.step(x, t, state)
51            self.step_target_prop(x, t, state[self, 'y'], state)
```

A.5 Feedback Alignment

Feedback Alignment makes use of a randomly generated weight matrix used in the backward pass that remains constant. This is a 3 layer fully-connected network implementation of feedback alignment.

Layer 1: Fully Connected FA Layer

1. Architecture: 32 output units, ReLU activation

2. Update: Gradient descent

3. X Update: N/A

Layer 2: Fully Connected FA Layer

1. Architecture: 10 output units

2. Update: Gradient Descent

3. X Update: Gradient descent using B

A.6 Direct Feedback Alignment

Direct Feedback Alignment makes use of a randomly generated weight matrix used in the backward pass that remains constant. In addition to that, each layer is directly connected to the output on the target pass. This is a 3 layer fully-connected network implementation of Direct Feedback Alignment.

Layer 1: Fully Connected DFA Layer

1. Architecture: 32 units Linear -> BatchNorm -> ReLU

2. Update: Cross Entropy Loss, 1e-3 LR

3. X Update: N/A

Layer 2: Fully Connected DFA Layer

1. Architecture: 32 units Linear -> BatchNorm -> ReLU

2. Update: Cross Entropy Loss, 1e-3 LR

3. X Update: N/A

Layer 3: Fully Connected FA Layer

1. Architecture: 10 units Linear

2. Update: Cross Entropy Loss, 1e-3 LR

3. X Update: N/A

Here is an example of code used for implementing Direct Feedback Alignment.

```
def DFALearner(LearningMachine):
    ...

    def __init__(self, in_features, hidden_features1, hidden_features2,
        out_features):

        super().__init__()
        self.layer3 = FALearner(nn.Linear(hidden_features2, out_features))
        self.layer2 = DFALearner(nn.Linear(hidden_features1, hidden_features2))
        self.layer1 = DFALearner(nn.Linear(in_features, hidden_features2))
        self.loss = ThLoss('cross_entropy')

    def assess_y(self, x, t, state, reduction_override: ) -> As
        return self.loss.assess_dict(x, t, reduction_override)

    def step(self, x: IO, t: IO, state: State):
        # go backward through the network.
        my_state = state.mine(self, x)
        # Here step is done before step_x. That means the update is executed first
        self.layer3.step(my_state.layer2, t)
        self.layer2.step(my_state.layer1, t)
        self.layer1.step(x, t)
        state[self, x, 'stepped'] = True

    # requires that step() is executed first
    @step_dep('stepped')
    def step_x(self, x: IO, t: IO, state: State) -> IO:
        # backpropgates from the first layer
        return self.layer1.step_x(x, state[self, x, t])
```

A.7 Convolutional - Decision

This is a three layer fully-connected network built from a convolutional network with one convolutional layer and a fully-connected layer followed by a decision tree classifier.

Layers 1 and 2: Convolutional neural network

1. Architecture Layer 1: (32 units, 4 kernel, 2 stride) -> BatchNorm -> ReLU
2. Architecture Layer 2: Linear -> BatchNorm
3. Update: MSE Loss, Adam Optim, Batch Size=128
4. X Update: N/A

Layer 2: Decision Tree Classifier

1. Architecture: 9 ensemble / 10 Max-depth 1 categorical output Decision Tree Classifier
2. Update: Scikit-learn defaults
3. X Update: Hill Climbing with Stochastic Incoming Layer