# ASSOCIATED LEARNING: A METHODOLOGY TO DECOMPOSE END-TO-END BACKPROPAGATION ON CNN, RNN, AND TRANSFORMER

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

We study associated learning (AL), an alternative methodology to end-to-end backpropagation (BP). We introduce the workflow to convert a neural network into a AL-form network such that AL can be used to learn parameters for various types of neural networks. We compare AL and BP on some of the most successful neural networks: convolutional neural networks, recurrent neural networks, and Transformers. Experimental results show that AL consistently outperforms BP on open datasets. We discuss possible reasons for AL's success, its limitations, and AL's newly discovered properties.

## 1 INTRODUCTION

Backpropagation (BP) is the keystone of modern deep learning. Although BP is the standard way to learn network parameters, it is far from ideal. Some of the most discussed issues of BP are optimization difficulties (e.g., vanishing gradients and exploding gradients (Hochreiter et al., 2001)) and training performance (e.g., backward locking (Jaderberg et al., 2017)).

It appears that custom network structures may be needed for different types of learning tasks. Among the various forms, convolutional neural networks (CNNs), recurrent neural networks (RNNs), and Transformer networks (along with their extensions, e.g., LSTM (Hochreiter & Schmidhuber, 1997) and VGG (Simonyan & Zisserman, 2015)) are particularly useful. These networks have been successfully applied in fields as varied as computer vision, natural language processing, signal processing, and others (Goodfellow et al., 2016; Deng & Yu, 2014).

This paper studies a new learning approach—associated learning (AL)—an alternative to end-to-end backpropagation learning. AL decomposes BP's global end-to-end training strategy into several small local optimization targets such that each component has an isolated gradient flow. However, since most components in AL do not interact with the final loss directly, we would expect AL-training models to be less accurate than BP. Surprisingly, the original AL paper compares AL and BP using the CNN network (and its extensions, e.g., VGG) and shows impressive results based on image classification datasets (MNIST, CIFAR-10, and CIFAR-100) (Kao & Chen, 2021). We continue this line of study in two ways. First, we discover more interesting properties of AL. Second, we apply AL on successful network structures, including VGG (Simonyan & Zisserman, 2015), LSTM (Hochreiter & Schmidhuber, 1997), and Transformer (Vaswani et al., 2017). Eventually, we compare the networks learned via AL and via BP on various tasks (image classification, sentiment analysis, and topic classification) based on public datasets (CIFAR-10, CIFAR-100, IMDB movie reviews, AG's News, Stanford Sentiment Treebank (SST), and DBpedia). We find that AL consistently outperforms BP in most datasets. Additionally, AL requires fewer epochs than BP when using early stopping but still yields excellent accuracy. These results suggest that AL is a strong alternative to BP, as AL is effective for various tasks.

The rest of the paper is organized as follows. In Section 2, we introduce associated learning and its properties. Section 3 presents the experimental results, including a comparison of the model accuracies and convergence speed of AL and BP on VGG, LSTM, and Transformer, a generalization test of AL and BP, and an ablation study. Section 4 reviews previous work on backpropagation alternatives. We discuss discoveries, limitations, and ongoing and future AL work in Section 5. Finally, we conclude our contribution in Section 6.

## 2 ASSOCIATED LEARNING

This section introduces associated learning and its properties.

### 2.1 AN OVERVIEW OF AL: AL FORM, TRAINING, AND INFERENCE

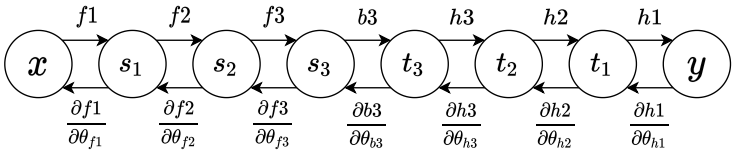


Figure 1: A six-layer neural network.

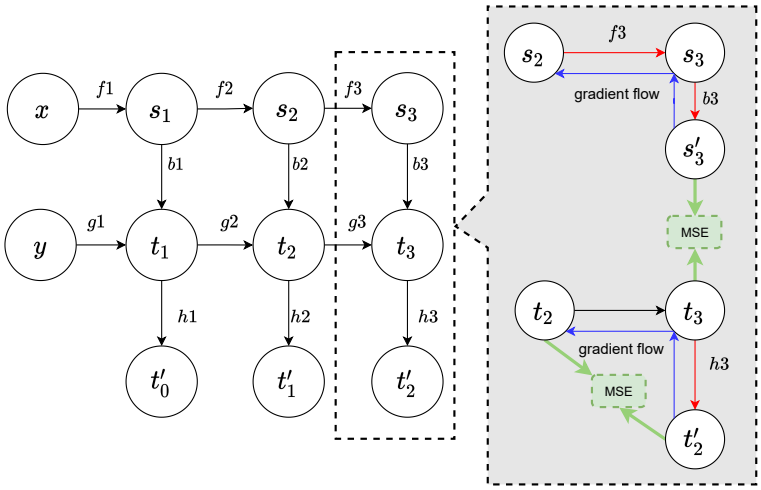


Figure 2: A six-layer neural network in the AL form required for AL learning. Red arrows: the inference path; green arrows: the objective function; blue arrows: gradient flow. We stop the gradient flow from an AL layer $i$ to its previous layer $i-1$ to ensure each layer is trained in isolation.

We show a typical multi-layer neural network with six hidden layers in Figure 1. Input $x$ goes through transformation functions $f_1, f_2, f_3, b_3, h_3, h_2$, and $h_1$ to obtain a prediction $\hat{y}$. The back-propagation algorithm updates the parameters in these functions such that the residual between $\hat{y}$ and a target $y$ decreases.

To apply AL, we must transform the network into a different structure, which we call *AL form* below. Specifically, instead of learning a function to map $x$ to $y$, the AL form learns the functions to transform $x$ to $x'$ and $y$ to $y'$ such that $x' \approx y'$ during the training phase. Algorithm **??** shows a standard format to convert a BP-based NN into AL form NN.

We use Figure 1 to illustrate how to transform a standard multi-layer neural network into its AL form. The transformation process includes the following steps.

For each layer in AL form, it propagate $x$ and $y$ jointly.

- Fold the original network in half (e.g., if given Figure 1 as the original network, the folded network is shown in the left of Figure 2). Each $(s_i, t_i)$ pair is called an *AL layer*.
- For each AL layer, we add two *forward functions* $f_i$ and $g_i$ to transform $s_{i-1}$ and $t_{i-1}$ into $s_i$ and $t_i$, respectively. In addition, we add another function $b_i$ (called a *bridge function*) to transform $s_i$ into $s'_i$ (assuming $s_0$ is $x$ and $t_0$ is $y$).
- For each AL layer, we add yet another function $h_i$ (called a decoding function) to transform $t_i$ into $t'_{i-1}$.

Take the network in Figure 2 as an example: let $x$ and $y$ be the input and target variables. Both $x$ and $y$ are used as the input of the AL-form network during AL training, and we use functions $f_i$ and $g_i$ ($i = 1, \ldots, 3$) to map $x$ and $y$ into $s_i$ and $t_i$, respectively. We then use bridge functions $b_i$ to transform $s_i$ into $s'_i$ such that (1) the shape of $s'_i$ is equivalent to the shape of $t_i$ and (2) the mean-squared error (MSE) between $s'_i$ and $t_i$ is minimized. We call the above objective *associated loss*, denoted as $\mathcal{L}_B$. By doing so, each pair of $f_i$ and $g_i$ is updated based on its associated loss. The behavior of aligning each $s'_i$ and $t_i$ in the preceding AL layers might seem not comprehensive at first glance, but we discover that this mechanism serves as more than just a measure of local update. By aligning $s'_i$ and $t_i$ layer by layer, the associated loss for deeper layers converges more easily.

So far, we have converted $x$ and $y$ into latent spaces such that they are close after the transformation. However, at the inference/prediction phase, we must transform $x$ into $y$. Unfortunately, based on the above mentioned forward functions and bridge functions, we can only transform $x$ to $t_3$ via the following path: $x \xrightarrow{f_1} s_1 \xrightarrow{f_2} s_2 \xrightarrow{f_3} s_3 \xrightarrow{b_3} t_3$; we cannot transform $t_3$ to $y$. Eventually, we use a decoding function $h_i$ for each AL layer to transform $t_i$ back to its previous state $t_{i-1}$. As a result, $g_i$ and $h_i$ together are regarded as the encoding and decoding functions of an autoencoder because $h_i(g_i(t_{i-1})) = t'_{i-1} \approx t_{i-1}$. We call the MSE between $t'_{i-1}$ and $t_{i-1}$ the *autoencoder loss*. Since $t'_{i-1} \approx t_{i-1}$ after training, $h_i$ can be an approximating function to convert $t_i$ into $t_{i-1}$. As a result, the inference path of the entire AL model is connected by

$$\hat{y} = (h_1 \circ h_2 \circ \ldots \circ h_\ell \circ b_\ell \circ f_\ell \circ \ldots \circ f_2 \circ f_1)(x), \tag{1}$$

where $\ell$ is the number of AL layers (e.g., $\ell = 3$ in Figure 2) and $\circ$ is the function composition operator.

We obtain the parameters for $f_i$, $g_i$, $b_i$, and $h_i$ by minimizing the following objective function:

$$\mathcal{L}_{AL} = \sum_{i=1}^{\ell} (\mathcal{L}_{A_i} + \mathcal{L}_{B_i}) = \sum_{i=1}^{\ell} \left( \left\| t'_{i-1} - t_{i-1} \right\|^2 + \left\| s'_i - t_i \right\|^2 \right), \tag{3}$$

where $\mathcal{L}_{B_i}$ and $\mathcal{L}_{A_i}$ are the associated loss and autoencoder loss of the $i$-th AL layer, respectively.

The inference and forward paths are identical in BP but differ in AL. Specifically, the forward path of AL layer $i$ consists of $f_i$, $b_i$, and $g_i$. However, the (full) inference path of AL is shown by Equation 1, which does not involve $g_i$.

In summary, during training, we make the latent representation of $x$ and $y$ closer to each other and, at the same time, train a decoder for each AL layer to transform the latent representation of $y$ back to its previous state. We stop the gradient flow from an AL layer $i$ to its previous layer $i - 1$ to ensure each layer is trained in isolation. For inference, an input $x$ goes forward through only $(f_1, \ldots f_\ell, b_\ell, h_\ell, \ldots, h_1)$; thus $(b_1, \ldots, b_{\ell-1}, g_1, \ldots, g_\ell)$ are not used for inference.

#### 2.1.1 Applying AL to Different Neural Network Structures

This section introduces how to convert some of the most successful neural network structures into their corresponding AL forms. We discuss vanilla CNN and VGG (used as the representative models for CNN-based models), LSTM (used as the representative model for RNN-based models), and Transformer. We also discuss how to integrate word embeddings into an AL-form network.

For all AL-form networks, functions $g_i$, $h_i$, and $b_i$ are formed by a linear transform followed by a non-linear activation function (e.g., tanh) in our experiment. Additionally, for simplicity, in this paper we apply only linear transformations as the bridge functions.

**CNN** For the bridge function in CNN architectures (here, vanilla CNN and VGG), we flatten its hidden representation as $s_i$ and send it to the bridge function. We follow the activation settings in Kao & Chen (2021) using sigmoid as the activation in $g_i$ and $h_i$, and ELU in $b_i$.

**RNN** For the bridge function in RNN architectures (here, LSTM and Bi-LSTM), since the input sequence has different lengths, we only use the representation of the last token as $s_i$. This is similar to sending the last token's hidden state to the classifier (as done in BP). We use tanh as the activation in $b_i$, $g_i$, and $h_i$.

**Transformer** In Transformer, we use mean pooling to aggregate all tokens' representations as $s_i$. We use tanh as the activation in $b_i, g_i$, and $h_i$.

**Word Embedding** Word embeddings are frequently used as the input of LSTM or Transformer for NLP tasks. We use mean pooling to aggregate all token's representations as $s_i$ and then use tanh as the activation in $b_i, g_i$, and $h_i$.

### 2.2 PROPERTIES OF AL

This section presents three properties of AL that do not exist in BP: forward shortcuts, dynamic layer accumulation, and pipelines.

#### 2.2.1 FORWARD SHORTCUTS

Forward shortcuts enable faster inference. Equation 1 shows the standard inference function for an AL-form network. However, we can also leverage "shortcut paths" for faster inference. Specifically, each bridge function $b_i$ can serve as a shortcut to transform $s_i$ to $s'_i$, which should be close to $t_i$ when the network is well trained. As a result, we can skip $f_j$, $b_j$, and $h_j$ for all $j > i$ to reduce the length of the inference function. In other words, we have multiple inference functions based on a single model. When inference time is critical, we can select a shorter inference path, e.g., $x \xrightarrow{f_1} s_1 \xrightarrow{b_1} t_1 \xrightarrow{h_1} y$. On the other hand, if inference time is unimportant, we can dynamically adjust the model complexity by modifying the number of AL layers used at the inference phase to reduce overfitting or underfitting.

#### 2.2.2 DYNAMIC LAYER ACCUMULATION

An AL-form network allows the dynamic creation of new layers during training. Specifically, we initially create an AL-form network with $k$ AL layers and train the network based on the strategy introduced in Section 2.1. If the model still underfits, we simply add the $(k+1)$-th AL layer on top of the first $k$ AL layers. We may choose to fix the parameters in the first $k$ AL layers and train only the parameters in the $(k+1)$-th AL layer. In contrast, when using a standard neural network, adding layers dynamically is more complicated.

#### 2.2.3 PIPELINE

In an AL-form network, the network parameters in different AL layers can be trained simultaneously via pipelines because each AL layer $i$ has a local objective function $\mathcal{L}_{A_i} + \mathcal{L}_{B_i}$. In contrast, it is difficult to simultaneously update the parameters of different layers for a standard BP-based neural network.

Given $n$ training instances in a standard neural network with $\ell$ layers, if we apply BP for training, the time complexity is $O(n\ell)$ because each instance must complete the entire forward ($\ell + 1$ transformations) and backward ($\ell + 1$ transformations) process. Thus, for such an architecture, the time complexity increases linearly with the total number of layers $\ell$. However, since each AL layer has its local objective, each layer can update its parameters without waiting for the gradient from preceding layers. As a result, for an AL-form network with $\ell/2$ AL layers, using $\ell/2$ computation units can make the parameter update process fully pipelined, as explained below.

We denote $task_i^j$ as the task of computing both forward and backward propagation for an AL layer $i$ based on the $j$-th mini-batch. If an AL layer $i$ finishes $task_i^1$ at time step $t$, at the next time step $t+1$, AL layers $i$ can continue performing $task_i^2$ since all parameters in this layer are updated based on $task_i^1$. Meanwhile, AL layer $i+1$ performs $task_{i+1}^1$. Consequently, the first mini-batch has a time complexity of $O(\frac{\ell}{2})$, and the following $n-1$ batches require only $O(1)$. The time complexity of one training epoch eventually reduces to $O(n+\ell)$.

# 3 EXPERIMENTS

To evaluate how AL works on different network architectures, we conducted experiments on image classification with vanilla CNN and VGG and on sentiment and topic classification with LSTM and Transformer (using pre-trained word embeddings as input). In addition, we conducted a case study on AL's generalization ability, evaluated by fitting AL models and BP models on different portions of noisy labels. Finally, we discuss possible reasons for AL's success based on ablation studies. We report more detailed settings in the Appendix.

## 3.1 DATASETS

We used four datasets (IMDB Review, AG's News corpus, DBpedia Ontology, and the Stanford Sentiment Treebank) to compare AL and BP on LSTM and Transformer, and two datasets (CIFAR-10 and Fashion-MNIST) to compare AL and BP on vanilla CNN and VGG.

The IMDB Review dataset (Maas et al., 2011) is a binary sentiment classification dataset containing 25k training data and 25k test data. AG's News corpus dataset (Zhang et al., 2015) includes four different news topics: world, sports, business, and sci/tech. It has 120k training data and 7.6k test data. The DBpedia Ontology classification dataset (Zhang et al., 2016) was constructed by picking 14 non-overlapping topic classes from DBpedia 2014, containing 560k training samples and 70k test samples. The Stanford Sentiment Treebank (SST2) dataset (Socher et al., 2013) is a binary sentiment classification dataset from GLUE (Wang et al., 2018). The dataset contains 67k training samples, 872 validation samples, and 1.8k test samples. The CIFAR-10 dataset contains color images from 10 classes; 50k are training samples, and 10k are test samples. Finally, the Fashion-MNIST dataset (Xiao et al., 2017) is a gray-scale image classification dataset with 10 classes. It consists of 60k training samples and 10k test samples.

## 3.2 TEXT CLASSIFICATION (LSTM AND TRANSFORMER)

We compared BP and AL on LSTM and Transformer based on the text classification task.

For LSTM, we used a 2-layer bidirectional LSTM with pre-trained GloVe word embeddings Pennington et al. (2014) as input. For Transformer, we used a 2-layer Transformer encoder, again with pre-trained GloVe word embeddings. During training, we recorded the accuracies of the validation set for every epoch and saved that model with the best performance on the validation set for testing. For each model, we repeated the process five times and report the average accuracy and standard deviation. For a fair comparison, we forced the AL parameter counts to not exceed those of the compared standard neural network by reducing AL's hidden dimension size. The accuracy results are given in Table 1. We also report the accuracies based on the predictions using shorter inference path in the same Table. In Table 2 and Table 3, we report the epoch that yielded the best results.

Table 1: Accuracy (mean $\pm$ standard deviation) of neural networks and corresponding AL versions on text classification datasets. Tran denotes Transformer, <X>-AL-full denotes full path inference on AL-form network <X>, and <X>-AL-SC$i$ denotes shortcut inference through bridge function $b_{i+1}$. EMB-AL-SC indicates inference through the shortcut in the embedding layer: after computing mean pooling on the word embeddings, the output is passed to a bridge function, followed by a decoder to obtain $y$.

| Method | IMDB | AGNews | SST | DBpedia |
|---|---|---|---|---|
| LSTM | 86.25 $\pm$ 0.63 | 90.32 $\pm$ 0.23 | **82.23** $\pm$ **0.93** | 97.34 $\pm$ 0.11 |
| LSTM-AL-full | 86.41 $\pm$ 0.61 | **91.53** $\pm$ **0.20** | 81.33 $\pm$ 0.31 | **98.30** $\pm$ **0.01** |
| LSTM-AL-SC1 | 86.16 $\pm$ 0.22 | 91.42 $\pm$ 0.17 | 80.83 $\pm$ 0.42 | 98.21 $\pm$ 0.06 |
| EMB-AL-SC | **87.80** $\pm$ **0.13** | 91.03 $\pm$ 0.28 | 80.57 $\pm$ 0.81 | 96.95 $\pm$ 0.03 |
| Tran | 83.45 $\pm$ .033 | 90.71 $\pm$ 0.28 | 76.48 $\pm$ 0.55 | 97.37 $\pm$ 0.06 |
| Tran-AL-full | 85.65 $\pm$ 0.77 | 91.17 $\pm$ 0.43 | **81.16** $\pm$ **0.11** | **97.55** $\pm$ **0.06** |
| Tran-AL-SC1 | 86.25 $\pm$ 1.34 | **91.49** $\pm$ **0.09** | 80.58 $\pm$ 0.67 | 96.53 $\pm$ 0.29 |
| EMB-AL-SC | **86.98** $\pm$ **0.72** | 91.43 $\pm$ 0.11 | 80.12 $\pm$ 0.57 | 97.14 $\pm$ 0.11 |

Table 2: Epoch (mean $\pm$ standard deviation) to reach the best accuracy on the validation set (smaller is better), and accuracy.

| Method | IMDB | | AGNews | |
|---|---|---|---|---|
| | Epoch | Accuracy | Epoch | Accuracy |
| LSTM | $15.4 \pm 2.60$ | $86.25 \pm 0.63$ | $25.4 \pm 13.42$ | $90.32 \pm 0.23$ |
| LSTM-AL | $\mathbf{4 \pm 1.22}$ | $\mathbf{86.41 \pm 0.61}$ | $\mathbf{2.4 \pm 0.89}$ | $\mathbf{91.53 \pm 0.20}$ |
| Tran | $35 \pm 6.63$ | $83.45 \pm 0.33$ | $\mathbf{3.75 \pm 0.96}$ | $90.71 \pm 0.28$ |
| Tran-AL | $\mathbf{4.4 \pm 0.55}$ | $\mathbf{85.65 \pm 0.77}$ | $20.25 \pm 12.39$ | $\mathbf{91.17 \pm 0.43}$ |

Table 3: Epoch (mean $\pm$ standard deviation) to reach best accuracy on validation set (smaller is better), and accuracy.

| Method | SST | | DBpedia | |
|---|---|---|---|---|
| | Epoch | Accuracy | Epoch | Accuracy |
| LSTM | $16 \pm 2.65$ | $\mathbf{82.33 \pm 0.93}$ | $31.8 \pm 5.26$ | $97.34 \pm 0.11$ |
| LSTM-AL | $\mathbf{2.25 \pm 1.89}$ | $81.33 \pm 0.31$ | $\mathbf{14.6 \pm 2.97}$ | $\mathbf{98.30 \pm 0.01}$ |
| Tran | $\mathbf{6 \pm 2.1}$ | $76.48 \pm 0.55$ | $\mathbf{6.75 \pm 1.71}$ | $97.37 \pm 0.06$ |
| Tran-AL | $9.67 \pm 0.58$ | $\mathbf{81.16 \pm 0.11}$ | $13.75 \pm 2.22$ | $\mathbf{97.55 \pm 0.06}$ |

Although AL is trained by segmented gradient flows, we found that AL-form LSTMs outperform their standard BP-trained counterparts on most datasets, similar to the results reported in Kao & Chen (2021): AL-trained CNNs usually exhibit improved performance. Also, we discover that LSTM-AL converges much faster than BP, which is a new discovery and suggests further improvements to overall training throughput if early stopping is applied. For Transformer, Transformer-AL also outperforms Transformer on every dataset. Also, we observe that standard Transformer easily overfits on smaller datasets like SST and IMDB, resulting in poor test accuracies, but Transformer-AL does not seem to have this weakness. Another discovery is that fast convergence for Transformer-AL seems only to appear on smaller datasets.

### 3.3 Image Classification (Vanilla CNN and VGG)

We also reproduced the experiments in Kao & Chen (2021) to determine whether AL-form CNNs also converge faster. In image classification, we report performance on the CIFAR-10 and Fashion-MNIST benchmarks. We selected FashionMNIST instead of MNIST because MNIST is likely too simple to show the power of different models: CIFAR-10 is more challenging than FashionMNIST since it contains three channels, whereas FashionMNIST is limited to grayscale images. Model details will be included in the Appendix. Each model was trained 5 times to calculate the mean and standard deviation of the accuracy. Table 4 shows the results: all AL models converge faster than BP. Also, AL models perform slightly better in terms of accuracy, consistent with the results demonstrated in Kao & Chen (2021).

Table 4: Accuracy (mean $\pm$ standard deviation) and epoch (mean $\pm$ standard deviation) to reach the best accuracy using different CNN models and their AL versions.

| | CIFAR-10 | | Fashion MNIST | |
|---|---|---|---|---|
| | Epoch | Accuracy | Epoch | Accuracy |
| CNN | $36.8 \pm 4.60$ | $82.64 \pm 0.61$ | $58.4 \pm 9.32$ | $91.57 \pm 0.11$ |
| CNN-AL | $\mathbf{33.6 \pm 4.97}$ | $\mathbf{85.16 \pm 0.11}$ | $\mathbf{11.6 \pm 1.67}$ | $\mathbf{92.07 \pm 0.18}$ |
| VGG | $47.2 \pm 7.82$ | $92.14 \pm 0.17$ | $32 \pm 4.34$ | $\mathbf{94.18 \pm 0.04}$ |
| VGG-AL | $\mathbf{35.6 \pm 2.19}$ | $\mathbf{92.48 \pm 0.15}$ | $\mathbf{25 \pm 3.83}$ | $93.80 \pm 0.08$ |

### 3.4 Generalization

To assess the generalizability of a model, a common approach is to empirically compare the errors on the training data and test data. Garg et al. (2021) show that a model is prone to overfitting if it

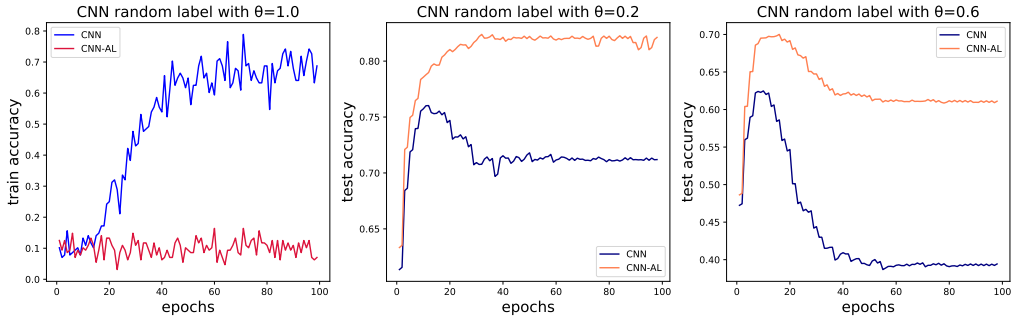


Figure 3: Training accuracy when $\theta = 1.0$ (left) and test accuracies when $\theta$ equals 0.2 (middle) and 0.6 (right) with different epochs.

closely fits a dataset with random labels. We use both methods to show that an AL-trained model may be more general than that trained by BP.

We took $\theta$ proportion of data, in which we changed the labels randomly. We used different proportions ($\theta = 0.1, 0.2, 0.3, 0.4, 0.6, 0.8, 1.0$) to train models. We report both the training accuracies and test accuracies on clean test data.

Table 5 shows the results. First, as shown in the last column (for a label noise rate of $\theta = 0$), the training accuracy of the CNN is larger than that of the CNN-AL, but CNN-AL outperforms CNN in terms of test accuracy. Second, with $\theta > 0$, it appears that the vanilla CNN network with BP training fits the randomly labeled training data much better than the AL-form CNN. However, if we evaluate model accuracies based on the clean test data, the AL-form CNN performs better. Taken together, the above empirical evidence suggests that the new model may generalize better (Garg et al., 2021). The relationship between accuracy and training epochs is given in Figure 3.

Table 5: Accuracies of different label noise rate $\theta$.

| | $\theta$ | 1.0 | 0.8 | 0.6 | 0.4 | 0.3 | 0.2 | 0.1 | 0.0 |
|---|---|---|---|---|---|---|---|---|---|
| Training | CNN | 68.75 | 64.84 | 82.03 | 93.75 | 89.84 | 96.88 | 96.09 | 97.48 |
| | CNN-AL | 8.59 | 41.41 | 50.00 | 65.63 | 74.22 | 78.13 | 85.94 | 96.26 |
| Testing | CNN | 9.96 | 23.25 | 39.23 | 56.30 | 63.60 | 71.18 | 77.04 | 82.64 |
| | CNN-AL | 10.00 | 38.23 | 64.74 | 76.77 | 78.70 | 82.19 | 83.40 | 85.16 |

### 3.5 Why AL works – an Ablation Study

We have empirically validated that AL generates excellent models, converges fast, and generalizes well. We further conducted ablation studies based on AG's news dataset to better understand why AL has these properties.

First, we ensure the gradient flow in each AL layer does not influence the gradients in other layers via the gradient stopping mechanism.[1] However, to optimize the objective function (Equation 3), allowing or blocking the gradient flow between AL layers are both acceptable. We sought to determine whether allowing gradient flow between AL layers improves model accuracy (although such a design may make pipelining infeasible). Hence, we disabled all the gradient stops (allowing gradients to flow through different AL layers). The results are shown in the first row of Table 6: accuracy is nearly unchanged. Therefore, given enough computing nodes, applying a pipelined AL increases training throughput without sacrificing model accuracy.

Second, to test the effect of associated loss, we disabled the gradient stops and also removed the associated loss (except the last AL layer) from the model. The results are shown in the second row of Table 6: even though the accuracy is similar to that with associated loss, far more epochs are needed to converge. This suggests that the bridge functions facilitate model training by gradually bringing $s'$ and $t$ closer.

[1]This is supported by `Tensor.detach()` in PyTorch and `tf.stop_gradient()` in TensorFlow.

With the third experiment, we sought to investigate the effect of the autoencoder components in the AL model. To do this, we passed the output of a normal Bi-LSTM to a bridge function, whose output $t_1$ was fed to the bottleneck layer of an autoencoder, which decodes $t_1$ back to the target $y$. Compared with the original Bi-LSTM classifier, the new design results in higher accuracy, as seen in the third (new design) and the fifth (LSTM) rows in Table 6. This suggests that replacing the standard classifier with a reverse AE $g_i, b_i$, and $h_i$ and training it with AL objectives could produce a better model.

Finally, for LSTM-AL, the inference path goes through 4 transformation functions $(b_3 \rightarrow h_3 \rightarrow h_2 \rightarrow h_1)$ after the LSTM module. To ensure a fair comparison, we also tested a network with a 3-layer MLP classifier after the standard LSTM model. The result is shown in the fourth row of Table 6. As shown, replacing the MLP from one layer (the 5th row of Table 6) to three layers does not produce a better model, i.e., the number of layers is not why LSTM-AL is a better model.

Table 6: Ablation studies on AG's News dataset to study why AL works.

| Method | Accuracy | Convergence epochs |
|---|---|---|
| LSTM-AL w/o grad. stops | $91.39 \pm 0.24$ | $2.2 \pm 0.45$ |
| LSTM-AL w/o $\mathcal{L}_B$ and w/o grad. stops | $91.48 \pm 0.15$ | $23.8 \pm 6.83$ |
| LSTM + RAE*1 | $90.75 \pm 0.13$ | $27.5 \pm 7.57$ |
| LSTM + MLP*3 | $90.15 \pm 0.15$ | $37.6 \pm 2.89$ |
| LSTM | $90.32 \pm 0.23$ | $25.4 \pm 13.42$ |
| LSTM-AL | $91.53 \pm 0.20$ | $2.4 \pm 0.89$ |

## 4 Related Work

Backpropagation is a fundamental component of deep learning. However, backpropagation suffers from optimization and performance issues. Related work usually concerns methodologies that better imitate the signal transmission process of biological neural networks, since biological neural networks are highly efficient in analyzing visual, audio, textual, and other signals (Lee et al., 2015; Kao & Chen, 2021; Nøkland, 2016; Ororbia et al., 2018; Ororbia & Mali, 2019). Alternatives to end-to-end backpropagation can be categorized into three types: proxy objective, target propagation, and synthetic gradients (Duan & Principe, 2021). Below we introduce each kind and the representative works.

The first type—proxy objective—uses a local alternative objective function for one layer $i$ and learns $\boldsymbol{\theta}_i$, the parameters for the current layer $i$. The model then fixes $\boldsymbol{\theta}_i$, adds one more layer $i+1$ with another local objective function, and learns $\boldsymbol{\theta}_{i+1}$, the parameters for the new layer. Consequently, proxy objective decouples the end-to-end backpropagation since the parameters in each layer are trained via a local objective function (Nøkland & Eidnes, 2019; Belilovsky et al., 2019; 2020).

The second type—target propagation—approximates $g_i$, the inverse of the downstream layers for every layer $i$. To update $\boldsymbol{\theta}_i$, the parameters of layer $i$, the model backpropagates the target (not gradient) through function $g_i$. Consequently, target propagation handles the situation where the relationship between the parameters and the cost is highly non-linear (e.g., discrete), which is otherwise difficult to solve using backpropagation (Lee et al., 2015; Meulemans et al., 2020).

The third type is synthetic gradients. This method learns a function to approximate the gradient for each layer and utilizes the approximated gradients as the true gradients to update the parameters. This method thus decouples the layer-wise parameters (Jaderberg et al., 2017; Czarnecki et al., 2017; Lansdell et al., 2019).

Associated learning is different from these three types of methods in several ways. First, both proxy gradient and target propagation propagate the signal (loss or target) from the output layer directly to each layer. Biologically, this could be implausible because when in deep networks, hidden layers near the inputs are unlikely to receive direct signals from the output layer that are far away. Associated learning, on the other hand, does not require each component to receive signals directly from the output layer. Additionally, although proxy gradient allows each layer to have a local objective, the parameters are still learned in a layer-wise fashion, so it may still be challenging

to learn the parameters in a pipelined fashion. As for synthetic gradients, empirical experiments show that gradients are difficult to predict, so model training based on synthetic gradients usually yields much worse accuracies than those for models trained via backpropagation.

## 5 DISCUSSION

Many of the current best practices and experiences (e.g., dropout, batch normalization, choice of activation functions, etc.) for training neural networks are tailored toward backpropagation training. In this paper, we follow most of these settings. However, since these policies are designed for BP, they may not be the best settings for AL. Consequently, it might also be appropriate to study preferred designs for AL, which is also a research direction of interest.

At first glance, the structure of an AL-form model may look similar to a ResNet, as the bridge functions in AL appear analogous to the residual connections in ResNet. Therefore, some may argue that AL's superiority is the result of its resemblance to ResNet. However, the residual connections in ResNet are implemented to mitigate accuracy saturation (He et al., 2016), and all the parameters are adjusted to fit a global objective function, whereas the bridge functions in AL (together with the forward functions and the decoding functions) are implemented to modularize the network such that the gradient flow in one AL layer does not pass to other layers. Therefore, the bridge functions and the residual connections are only structurally similar; their purposes and functionalities are entirely different.

An AL-form network may also appear similar to contrastive learning (CL). Particularly, CL transforms a pair of images $i$ and $j$ (along with their augmented images $i'$ and $j'$) into vectors with large distances between (1) images $i$ and $j$ and (2) augmented images $i'$ and $j'$, with small distances between (1) an image $i$ and its augmented image $i'$ and (2) an image $j$ and its augmented image $j'$. This process is related to AL because both models convert inputs to vectors and compare the distance between the transformed vectors. Following this line, we are interested in studying the relationship between AL and CL in two ways. First, we could apply AL to CL such that CL is trained more effectively. Second, we could extend the AL model such that the inputs include a pair of samples $x_1$ and $x_2$ with different labels and design an objective function that maximizes the distance between the vectors transformed from $x_1$ and $x_2$ (since their labels are different) and simultaneously minimizes the current AL loss function (Equation 3).

## 6 CONCLUSION

This paper studies associated learning, an alternative methodology to end-to-end backpropagation. Since backpropagation is the cornerstone of today's deep learning but still suffers from optimization and training performance issues, it is appropriate to study alternative methodologies to backpropagation. We discussed unique properties of AL that BP does not have. AL provides multiple inference paths; it allows dynamic layer accumulation during training; it supports pipeline training, and, perhaps most importantly, its prediction power is comparable to and frequently better than backpropagation on many of the most successful neural network structures. Ablation studies disclose why AL may work and demonstrate AL's effectiveness and architecture flexibility in training and testing.

## REFERENCES

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.

Eugene Belilovsky, Michael Eickenberg, and Edouard Oyallon. Greedy layerwise learning can scale to ImageNet. In *International Conference on Machine Learning*, pp. 583–593. PMLR, 2019.

Eugene Belilovsky, Michael Eickenberg, and Edouard Oyallon. Decoupled greedy learning of CNNs. In *International Conference on Machine Learning*, pp. 736–745. PMLR, 2020.

Wojciech Marian Czarnecki, Grzegorz Świrszcz, Max Jaderberg, Simon Osindero, Oriol Vinyals, and Koray Kavukcuoglu. Understanding synthetic gradients and decoupled neural interfaces. In *International Conference on Machine Learning*, pp. 904–912. PMLR, 2017.

Li Deng and Dong Yu. Deep learning: Methods and applications. *Foundations and Trends in Signal Processing*, 7(3–4):197–387, 2014.

Shiyu Duan and Jose C Principe. Training deep architectures without end-to-end backpropagation: A brief survey. *arXiv preprint arXiv:2101.03419*, 2021.

Saurabh Garg, Sivaraman Balakrishnan, J Zico Kolter, and Zachary C Lipton. RATT: Leveraging unlabeled data to guarantee generalization. *arXiv preprint arXiv:2105.00303*, 2021.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT press, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8): 1735–1780, 1997.

Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. Gradient flow in recurrent nets: The difficulty of learning long-term dependencies, 2001.

Max Jaderberg, Wojciech Marian Czarnecki, Simon Osindero, Oriol Vinyals, Alex Graves, David Silver, and Koray Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. In *International Conference on Machine Learning*, pp. 1627–1635. PMLR, 2017.

Yu-Wei Kao and Hung-Hsuan Chen. Associated learning: Decomposing end-to-end backpropagation based on autoencoders and target propagation. *Neural Computation*, 33(1):174–193, 2021.

Benjamin James Lansdell, Prashanth Ravi Prakash, and Konrad Paul Kording. Learning to solve the credit assignment problem. *arXiv preprint arXiv:1906.00889*, 2019.

Dong-Hyun Lee, Saizheng Zhang, Asja Fischer, and Yoshua Bengio. Difference target propagation. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 498–515. Springer, 2015.

Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pp. 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P11-1015`.

Alexander Meulemans, Francesco S Carzaniga, Johan AK Suykens, João Sacramento, and Benjamin F Grewe. A theoretical framework for target propagation. *arXiv preprint arXiv:2006.14331*, 2020.

Arild Nøkland. Direct feedback alignment provides learning in deep neural networks. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pp. 1037–1045, 2016.

Arild Nøkland and Lars Hiller Eidnes. Training neural networks with local error signals. In *International Conference on Machine Learning*, pp. 4839–4850. PMLR, 2019.

Alexander G Ororbia and Ankur Mali. Biologically motivated algorithms for propagating local target representations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 4651–4658, 2019.

Alexander G Ororbia, Ankur Mali, Daniel Kifer, and C Lee Giles. Conducting credit assignment by aligning local representations. *arXiv preprint arXiv:1803.01834*, 2018.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary 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. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1532–1543, Doha, Qatar, October 2014. Association for Computational Linguistics. doi: 10.3115/v1/D14-1162. URL https://aclanthology.org/D14-1162.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pp. 1631–1642, Seattle, Washington, USA, October 2013. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/D13-1170.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pp. 5998–6008, 2017.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms, 2017.

Xiang Zhang, Junbo Jake Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *NIPS*, 2015.

Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification, 2016.

# 7 APPENDIX

## 7.1 IMPLEMENTATION ENVIRONMENT DETAILS

For LSTM and Transformer, we used the PyTorch Paszke et al. (2019) package. For CNN, we used the Tensorflow Abadi et al. (2015) package to build the AL models (vanilla CNN and a visual geometry group (VGG) network (Deng & Yu, 2014)). We trained the models with GeForce 3090 GPUs on Ubuntu Linux 20.04. Since current hardware development does not support actual pipeline training, we simulated pipeline training by blocking gradient flows between AL layers with `Tensor.detach()` in PyTorch and with `tf.stop_gradient()` in TensorFlow.

## 7.2 TEXT CLASSIFICATION

During training, the most frequent 30k words in the training corpus were included in the word embedding. We replaced the remaining words with the [UNK] token. We trained all text classification experiments with 50 epochs. We used the Adam optimizer.

### 7.2.1 BI-LSTM

For Bi-LSTM, we applied gradient clipping with the value 5. For the standard version of Bi-LSTM, we experimented with learning rates of 0.0005 and 0.001, and report that with better accuracy: we used 0.0005 for AG's news and IMDB, and 0.001 for DBpedia and SST. We also adjusted the learning rate for AL, but the learning rate has little influence on the accuracy and the convergence speed. We used a batch size of 64 for all Bi-LSTM experiments. For IMDB and DBpedia, we set the max sequence length to 500. For AG's News and SST, we set the max sequence length to the longest sequence length; the exact length settings were applied to the Transformer experiments. The objective of the standard Bi-LSTM was cross-entropy loss. Details are given in Table 7.

Table 7: Bi-LSTM settings.

| | emb dim | lstm1 dim | lstm2 dim | $g$ dim | $h$ dim | lr |
|---|---|---|---|---|---|---|
| LSTM | 300 | 350 | 350 | – | – | 0.0005 |
| LSTM-AL | 300 | 300 | 300 | 128 | 128 | 0.001 |

### 7.2.2 TRANSFORMER

For the standard Transformer model, we used negative log-likelihood as the objective with multi-head attention (with 6 heads). We set the batch size to 128 because the standard Transformer converges only with a large batch size in our experiment. See Table 8.

Table 8: Transformer settings.

| | emb dim | layer1 dim | layer2 dim | head | $g$ dim | $h$ dim | lr |
|---|---|---|---|---|---|---|---|
| Tran | 300 | 512 | 512 | 6 | – | – | 0.00025 |
| Tran-AL | 300 | 256 | 256 | 6 | 128 | 128 | 0.00025 |

## 7.3 IMAGE CLASSIFICATION

We normalized the image pixels for both FashionMNIST and CIFAR-10 datasets as a preprocessing step. Since CIFAR-10 is more challenging, we also performed data augmentation on this dataset by resizing, random cropping, random flipping, and adjusting the brightness of the images. We set the batch size to 128. We set the initial learning rate to 0.0001 and applied scheduled decay until converging. We used Adam as the optimizer and cross-entropy loss for the standard version.

**Vanilla CNN Model** We implemented the vanilla CNN by modifying the previous code. The CNN contains 13 hidden layers and an output layer. The first 8 hidden layers are convolution layers; the last is flattened to a fully connected layer with 1280 neurons. The subsequent layers consist of 4

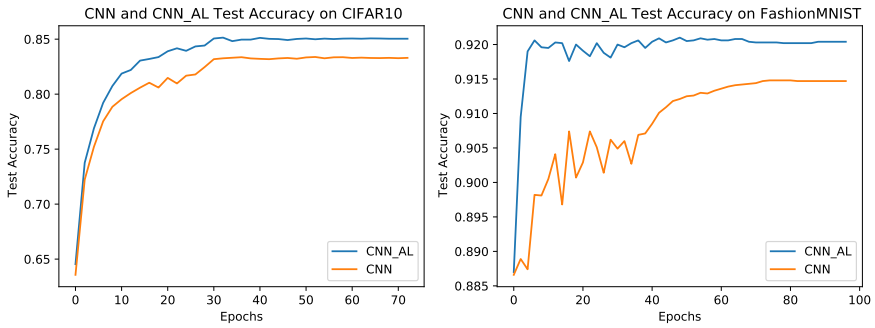


Figure 4: Testing accuracy vs. epochs for CNN and CNN-AL on CIFAR-10 and FashionMNIST.

feedforward layers, followed by an output layer with 10 classes. We set the initial learning rate to 0.0001 and applied scheduled decay.

## 8 PYTORCH PSEUDOCODE

Here we provide a simple pytorch pseudocode of a 5 layer AL network.

```
import torch
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self, d1, d2):
        super().__init__()
        self.f = nn.Linear(d1,d2)
        self.act = nn.Sigmoid()

    def forward(self, x):
        return self.act(self.f(x))

class AL(nn.Module):
    def __init__(self, class_num):
        super().__init__()

        self.f = nn.ModuleList([MLP(10, 10) for i in range(5)])
        self.g = nn.ModuleList([MLP(10, 10) for i in range(5)])
        self.b = nn.ModuleList([MLP(10, 10) for i in range(5)])
        self.h = nn.ModuleList([MLP(10, 10) for i in range(5)])
        self.g[0], self.h[0] = MLP(class_num, 10), MLP(10, class_num)

        self.mse = nn.MSELoss()
        self.c = class_num

    def forward(self, x, y):

        loss = 0.0
        s_i, t_i = x, nn.functional.one_hot(y, num_classes=self.c).float()
        # convert y into one-hot vector for linear input
        for i in range(5):

            last_t_i = t_i
            s_i = self.f[i](s_i)
            s_i_prime = self.b[i](s_i)
            t_i = self.g[i](t_i)
            t_i_prime = self.h[i](t_i)

            loss += self.mse(t_i.detach(), s_i_prime) +
                self.mse(last_t_i, t_i_prime)
            s_i, t_i = s_i.detach(), t_i.detach()

        return loss

    def inference(self, x):
        for layer in self.f:
            x = layer(x)
        y = self.b[-1](x)
        for layer in reversed(self.h):
            y = layer(y)
        return y

model = AL(5)
x, y = torch.rand(30, 10, dtype=torch.float), torch.randint(high=4, size=(30, ))
out, pred = model(x, y), model.inference(x)
```