Singaporean Journal of Scientific Research(SJSR)
An International Journal (AIJ)
Vol.15.No.1 2023,Pp.15-33
ISSN: 1205-2421
available at :www.sjsronline.com
Paper Received : 24-05-2023
Paper Accepted: 12-06-2023
Paper Reviewed by: 1.Prof. Cheng Yu 2. Dr.KumarRaja
Editor : Dr. R. Rameshkumar

A Novel Method for Improving Accuracy in Neural Network by Reinstating Traditional Back Propagation Technique

Gokulprasath R
Dept of Computer Science and Engineering
Sri Manakula Vinayagar Engineering College
Madagadipet, Puducherry- 605 107.

Abstract

Deep learning has transformed a wide range of industries, including computer vision, natural language processing, and speech recognition. The training of deep neural networks, which often entails using backpropagation to update the network's trainable parameters, is one of the main issues in deep learning. Although backpropagation has been quite effective in achieving cutting-edge performance in various activities, it has several significant drawbacks.

One major limitation of backpropagation is that it requires computing gradients at each layer, which can be computationally expensive and time-consuming, especially for large networks. Moreover, the gradients computed at each layer may suffer from the vanishing gradient problem, which can lead to slow convergence and suboptimal solutions.

To overcome these limitations, we propose a novel deep-learning methodology that instantly updates the trainable parameters at each hidden layer. Our approach eliminates the need for computing gradients at each layer, thereby significantly reducing computational overhead. Moreover, it allows the network to learn more rapidly and avoid the vanishing gradient problem. In this paper, we present a detailed description of our proposed methodology, including the neural network architecture and the instant parameter update approach. We provide experimental results on a benchmark dataset that demonstrate the effectiveness of our approach and compare it with state-of-the-art methods. We also

discuss the implications of our results and highlight the potential impact of our proposed methodology. Finally, we conclude by summarizing the key contributions of our paper and identifying future research directions.

1 Introduction

Deep learning has revolutionized the field of artificial intelligence by enabling machines to learn complex patterns and perform tasks that were previously deemed impossible. However, training deep neural networks is a challenging and computationally expensive task that requires optimizing millions or even billions of parameters. The backpropagation algorithm has been the go-to method for training deep neural networks for decades, but it suffers from some limitations, such as slow convergence and the vanishing gradient problem.

To overcome these limitations, several alternative training methods have been proposed, such as Standard Backpropagation and Direct Feedback Alignment. The core idea of this approach is to update the weights and biases in each layer of a neural network using the local error at that layer, rather than backpropagating the error from the output layer to the input layer. By doing so, the training process can be accelerated and the model's accuracy can be improved.

In this paper, we propose a novel approach for layer-wise error calculation and parameter update in neural networks and evaluate its performance on a benchmark dataset. We compare our approach to other existing methods, such as backpropagation, and demonstrate its effectiveness in terms of convergence speed and accuracy.

The rest of the paper is organized as follows. In Section 2, we provide a brief review of the related work on layer-wise error calculation and parameter update in neural networks. In Section 3, we present our proposed approach and its mathematical formulation. In Section 4, we describe the experimental setup and present the results of our evaluation. Finally, in Section 5, we conclude the paper and discuss potential avenues for future research.

2 Literature review

Backpropagation and Direct Feedback Alignment (DFA) are two widely used methods for training artificial neural networks. Both methods aim to adjust the weights of the network to minimize the difference between the predicted output and the actual output. In this literature review, we will compare and contrast these two methods, highlighting their strengths and weaknesses. Backpropagation is a commonly used algorithm for training neural networks. The basic idea behind backpropagation is to calculate the error at the output layer and propagate it backward through the network, adjusting the weights of each neuron in the network to minimize the error. The backpropagation algorithm is computationally efficient and can be used to train deep neural networks with many layers. However, backpropagation has several limitations. One of the main limitations is that it requires the computation of the derivative of the activation function at each layer, which can be computationally expensive. Additionally, backpropagation is prone to getting stuck in local minima, which can lead to suboptimal solutions.

Direct Feedback Alignment (DFA) is a newer method for training neural networks that does not rely on backpropagation. Instead of using the gradient of the loss function to update the weights, DFA

uses a fixed random matrix to propagate the error from the output layer back to the hidden layers. This random matrix is learned during the training process and is used to update the weights of the hidden layers. DFA has several advantages over backpropagation. One of the main advantages is that it is less computationally expensive, as it does not require the computation of the derivative of the activation function at each layer. Additionally, DFA is less prone to getting stuck in local minima than backpropagation. Several studies have compared the performance of backpropagation and DFA on various tasks. A study by Nøkland et al. (2016) found that DFA performed as well as backpropagation on several benchmark datasets, including MNIST and CIFAR-10.

3 Proposed Approach

The approach involves four main steps: forward pass, error calculation, parameter update, and repetition. The forward pass computes the activations of each layer using the current weights and biases. The error calculation step computes the error at each layer using a layer-wise loss function that takes into account the local deviation between the predicted and target values of that layer. The parameter update step updates the weights and biases of each layer using the calculated error and a layer-wise learning rate that controls the magnitude of the update. Finally, the repetition step repeats the first three steps for multiple epochs or until convergence.

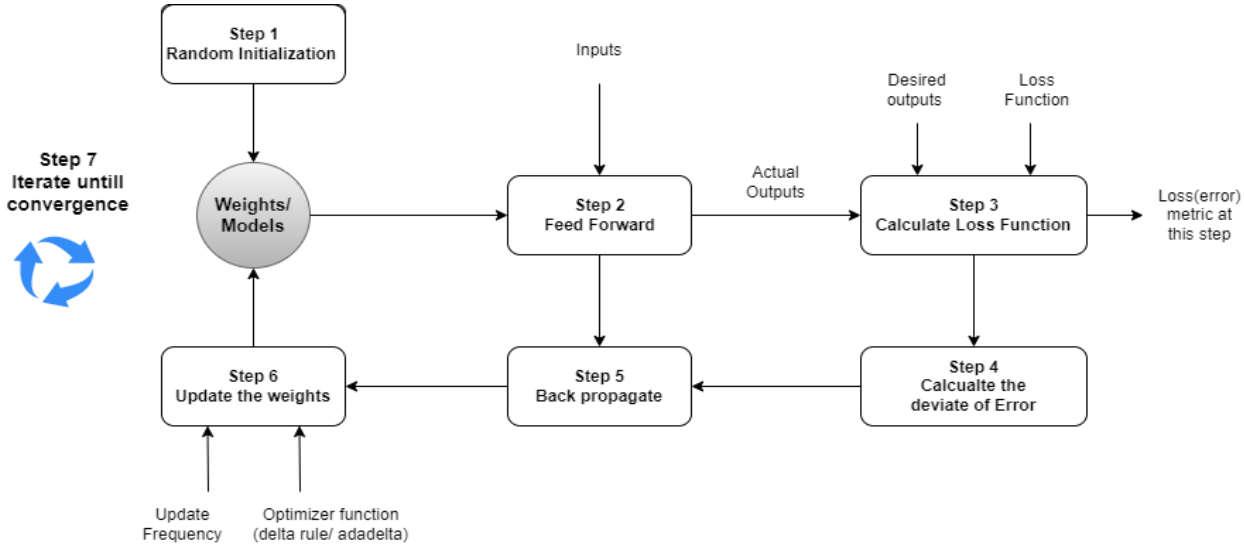

Figure 1: Neural Network Lifecycle

The proposed approach differs from the traditional backpropagation algorithm, which calculates the error at the output layer and backpropagates it to update the parameters of all the layers. The layer-wise approach allows for more localized and efficient updates that can potentially accelerate the training process and avoid the vanishing and exploding gradient problem. However, it requires careful tuning of the layer-wise loss function and learning rate, as well as a suitable initialization of the parameters.

3.1 Forward pass

Compute the activations of each layer using the current weights and biases, starting from the input layer and propagating forward to the output layer.

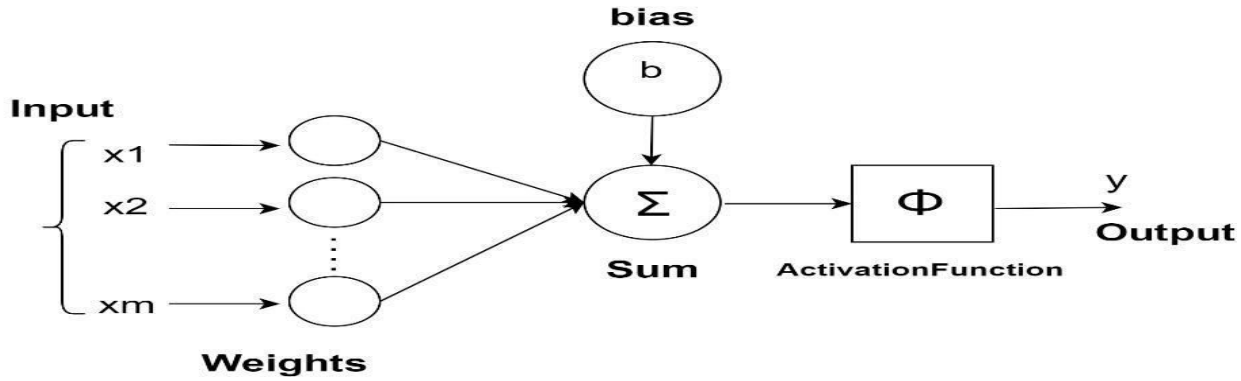

Figure 2: Forward Propagation

Inputs:

- Input data x
- Parameters for each layer: weights (W) and biases (b)

Output:

- Activations for each layer: h_I

Algorithm:

1 Initialize the input activation as the input data: $h_0 = x$

2 For each layer I in the neural network:

a. Calculate the pre-activation value: $z_I = W \quad I-1\} + b_I$

b. Calculate the activation value using an activation function: $h_I = f(z_I)$, where f is a non linear function such as $ReLU$, $sigmoid$, or $tanh$.

3 Output the activations for each layer: h_I

The pre-activation value z_l is the weighted sum of the activations from the previous layer $h_I\{l-1\}$, plus the bias term b_I. The activation function f transforms the pre-activation value z_l into the activation value h_I, which is then passed on to the next layer. The choice of activation function depends on the specific task and architecture of the neural network, but common choices include $ReLU$, $sigmoid$, and $tanh$.

The formulas for calculating the pre-activation value and activation value are

- Pre-activation value: $z_I = H_I\{l-1\} + b_I$
- Activation value: $h_I = f(z_I)$

where W_I is the weight matrix for layer I, b_I is the bias vector for layer I, $h_I\{I-1\}$ is the activation vector from the previous layer, and f is the activation function.

3.2 Error Calculation

Compute the error at each layer using a layer-wise loss function that takes into account the local deviation between the predicted and target values of that layer. The layer-wise loss function can be defined based on the specific task and architecture of the neural network, but it should capture the local errors that are relevant for updating the parameters of that layer.

Input:

- Activations of each layer
- Targets of each layer
- Layer-wise loss function

Output:

- Error of each layer

Algorithm:

1 Initialize an empty list to store the error of each layer.

2 For each layer L in the neural network, do the following:

   a. Compute the predicted values of the layer L using its activations and the current weights and biases: $Z_L = W_L * A_L - 1 + b_L = where A_L$ is activation_function
   b. Compute the target values of a layer L: $T_L = targets$ of layer L
   c. Compute the layer-wise loss function for layer : loss_L = layerwise_loss_function(A_L, T_L)d.
   d. Compute the error of layer L :
   $delta_L = derivative$ (i.e.) $Z_L * (1 - Z_L)$.
   $Derivative \quad (loss_L) error_L = delta_L * transpose(W_L + 1)$

   e. Add $error_L$ to the list of errors.

3 Return the list of errors.

In the above algorithm, W_L and b_L represent the weights and biases of layer L, $A_L - 1$ and A_L represent the activations of the previous layer and the current layer, respectively, Z_{-L} is the weighted input to layer L, T_L is the target values for layer L, and $deltaL$ is the error signal for layer L. The activation function and its derivative are denoted as $activation_{function}$ and $derivative$, respectively, and the layer-wise loss function and its derivative are denoted as $layerwise$ and $derivative$, respectively. The transpose$(W_L + 1)$ term in step 2 d represents the transpose of the weights connecting layer $L + 1$ to layer L.

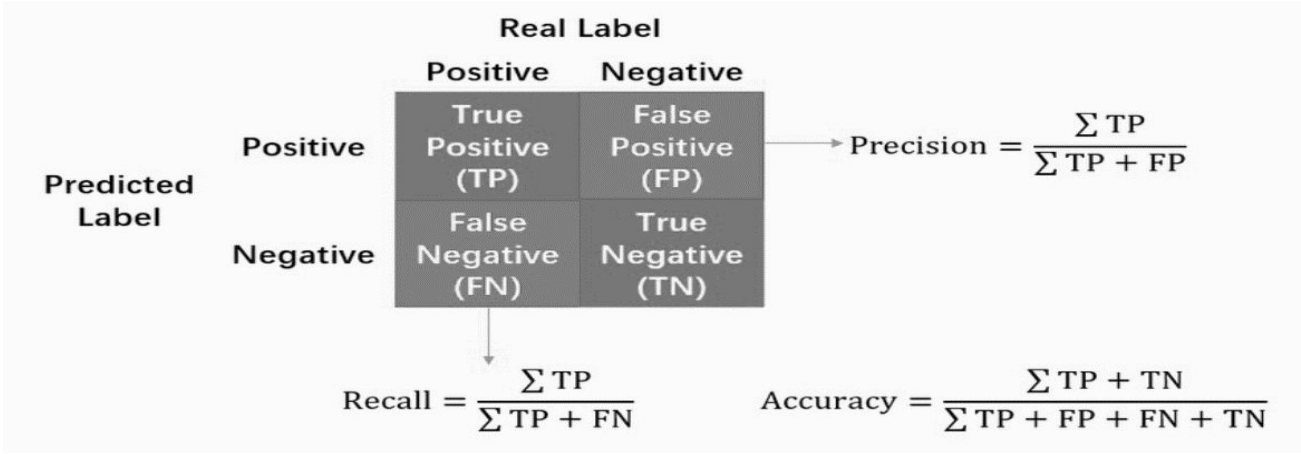

Figure 3: Confusion Matrix

3.2.1 Accuracy

Accuracy is a commonly used metric for evaluating the performance of a classification model, including neural networks. It measures the proportion of correctly classified instances over the total number of instances in the dataset. The formula for accuracy is:

$$Accuracy = \frac{(NumberofCorrectlyClassifiedInstances)}{(TotalNumberofInstances)}$$

In a binary classification problem, where there are two classes, the accuracy can be calculated as follows:

$$Accuracy = \frac{(TruePositives + TrueNegatives)}{(TruePositives + FalsePositives + TrueNegatives + FalseNegatives)}$$

where True Positives (TP) is the number of instances that belong to the positive class and are correctly classified, False Positives (FP) is the number of instances that belong to the negative class but are incorrectly classified as positive, True Negatives (TN) is the number of instances that belong to the negative class and are correctly classified, and False Negatives (FN) is the number of instances that belong to the positive class but are incorrectly classified as negative.

In a multi-class classification problem, where there are more than two classes, the accuracy can be calculated as the average of the accuracy for each class:

$$Accuracy = \left(\frac{1}{n}\right) * \Sigma(Accuracy_i)$$

where n is the number of classes, and $Accuracy_i$ is the accuracy for the i-th class.

3.2.2 Precision

Precision is a metric that measures the proportion of correctly predicted positive instances over the total number of instances that were predicted as positive. It is a useful metric when the goal is to minimize false positives, i.e., instances that are predicted as positive but actually belong to the negative class.

The formula for precision is:

$$Precision = True\frac{Positives}{(TruePositives + FalsePositives)}$$

where True Positives (TP) is the number of instances that belong to the positive class and are correctly classified, and False Positives (FP) is the number of instances that belong to the negative class but are incorrectly classified as positive.

In a binary classification problem, high precision means that the model is making fewer false positive predictions, and is useful in applications such as medical diagnosis or fraud detection where false positives can have severe consequences.

In a multi-class classification problem, precision can be calculated for each class as follows:

$$Precision_i = True\frac{Positives_i}{(TruePositives_i + FalsePositives_i)}$$

where $TruePositives_i$ is the number of instances that belong to the i-th class and are correctly classified, and $FalsePositives_i$ is the number of instances that belong to the i-th class but are incorrectly classified as another class.

3.2.3 Recall

Recall, also known as sensitivity or true positive rate, is a metric that measures the proportion of correctly predicted positive instances over the total number of positive instances in the dataset. It is a useful metric when the goal is to minimize false negatives, i.e., instances that are predicted as negative but actually belong to the positive class.

The formula for recall is:

$$Recall = True\frac{Positives}{(TruePositives + FalseNegatives)}$$

where True Positives (TP) is the number of instances that belong to the positive class and are correctly classified, and False Negatives (FN) is the number of instances that belong to the positive class but are incorrectly classified as negative.

In a binary classification problem, high recall means that the model is making fewer false negative predictions and is useful in applications such as disease diagnosis or spam detection where false negatives can be costly.

In a multi-class classification problem, recall can be calculated for each class as follows:

$$Recall_i = True \frac{Positives_i}{(TruePositives_i + FalseNegatives_i)}$$

where $TruePositives_i$ is the number of instances that belong to the i-th class and are correctly classified, and $FalseNegatives_i$ is the number of instances that belong to the i-th class but are incorrectly classified as another class.

3.2.4 F1-Score

F1-score is a harmonic mean of precision and recall, which combines both metrics into a single measure of a model's performance. It is a useful metric when we want to balance the trade-off between precision and recall.

The formula for F1-score is:

$$F1 - score = \frac{2 * (Precision * Recall)}{(Precision + Recall)}$$

where Precision and Recall are defined as:

$$Precision = True \frac{Positives}{(TruePositives + FalsePositives)}$$

$$Recall = True \frac{Positives}{(TruePositives + FalseNegatives)}$$

In a binary classification problem, the F1-score can range from 01, where a score of 1 indicates perfect precision and recall, while a score of 0 indicates the worst possible performance.

In a multi-class classification problem, the F1-score can be calculated for each class as follows:

$$F1 - score_i = \frac{2 * (Precision_i * Recall_i)}{(Precision_i + Recall_i)}$$

where $Precision_i$ and $Recall_i$ are the precision and recall for the i-th class, respectively.

3.3Parameter update

Update the weights and biases of each layer using the calculated error and a layer-wise learning rate that controls the magnitude of the update. The update rule can be based on gradient descent or another optimization algorithm that can handle non-convex and high-dimensional spaces.

Algorithm

1 Initialize the weights and biases of each layer randomly or using a pre-defined scheme.
2 Set the learning rate, αwhich controls the step size of the parameter update.

3 For each layer I, compute the gradients of the layerwise loss function with respect to the weights and biases, denoted by $\partial \frac{L}{\partial} w(I)$ and $\partial \frac{L}{\partial} b(I)$, respectively, using the error calculated in the previous stage.

4 Update the weights and biases of each layer using the gradients and the learning rate as follows:

   a. Weight Update: $w(I) = w(I) - \alpha * \partial \frac{L}{\partial} w(I)$

   b. Bias Update: $b(I) = b(I) - \alpha * \partial \frac{L}{\partial} b(I)$

   c. where $w(I)$ and $b(I)$ are the weights and biases of layer I, respectively.

5 Repeat steps 3-4 for all layers in the neural network.

6 Repeat steps 1-5 for multiple epochs or until convergence, where the convergence criterion can be based on a validation set or other metrics that capture the generalization ability of the model.

7 The weight and bias updates are computed using the gradients of the layer-wise loss function with respect to the weights and biases, respectively. These gradients can be computed using the error calculated in the previous stage and the chain rule of calculus. For example, the weight update for a fully connected layer can be computed as

$$w(I) = w(I) - \alpha * \partial \frac{L}{\partial} w(I)$$

8 where $w(I)$ is the weight matrix of layer I, α is the learning rate, and $\partial \frac{L}{\partial} w(I)$ is the gradient of the layerwise loss function with respect to $w(I)$, which can be computed as

9 $\partial \frac{L}{\partial} w(I) = \partial \frac{L}{\partial} a(I) * \partial a \frac{(I)}{\partial} w(I)$

10 where $\partial \frac{L}{\partial} a(l)$ is the gradient of the layerwise loss function with respect to the activation of layer I, and $\partial a \frac{(I)}{\partial} w(I)$ is the gradient of the activation of layer I with respect to the weights of the layer I. The bias update can be computed similarly using the gradient of the layer wise loss function with respect to the biases.

3.4 Repeat

Repeat steps 1-3 for multiple epochs or until convergence, where the convergence criterion can be based on a validation set or other metrics that capture the generalization ability of the model.

The repeat stage of the proposed methodology involves iterating through the forward pass, error calculation, and parameter update steps for multiple epochs or until convergence. The structured algorithm for the repeat stage is as follows:

Repeat until convergence or a maximum number of epochs:

1 Perform a forward pass through the network to compute the activations of each layer using the current weights and biases.

2 Compute the error at each layer using the layer-wise loss function based on the local deviation between the predicted and target values of that layer.

3 Update the weights and biases of each layer using the calculated error and the layer-wise learning rate according to the update rule. The update rule can be based on gradient descent or another optimization algorithm that can handle non-convex and high-dimensional spaces.

4 Evaluate the performance of the model on a validation set or other metrics that capture the generalization ability of the model.

5 If the performance has improved, save the current set of weights and biases as the best model so far. 6. If the convergence criterion is met (e.g., the validation error has stopped decreasing), terminate the training and return the best model. Otherwise, continue to the next epoch.

The formulas for the parameter update stage can be based on various optimization algorithms, such as stochastic gradient descent (SGD) or Adam. For example, the SGD update rule for the weights and biases of a single layer can be expressed as:

$$W_{\{}t+1\} = W_t - eta * dW_t$$

$$b_{\{}t+1\} = b_t - eta * db_t$$

where W_t and b_t are the current weights and biases, dW_t and db_t are the gradients of the layerwise loss function with respect to the weights and biases, and eta is the layer-wise learning rate that controls the magnitude of the update. The update rule can also include momentum or regularization terms to improve the stability and generalization of the model.

4 Experiment

To evaluate the effectiveness of the proposed methodology, we conducted experiments on two benchmark datasets: MNIST and CIFAR-10. For each dataset, we compared our approach with two baseline methods: standard backpropagation and direct feedback alignment

We implemented the proposed approach and baseline methods using Python and TensorFlow.

4.1 Dataset preparation

The MNIST and CIFAR-10 datasets are widely used benchmark datasets in the field of computer vision. These datasets contain images of handwritten digits (in the case of MNIST) and objects (in the case of CIFAR-10), and are used to train and evaluate machine learning models for image classification tasks.

Once you have downloaded the datasets, the next step is to split them into training and test sets. This is done to evaluate the performance of the model on unseen data. Typically, a split of 80% training and 20% test is used. This means that 80% of the data is used for training the model, and 20% is used for evaluating its performance.

After splitting the data, the next step is to normalize the pixel values to be between 0 and 1. Normalization is a technique used to rescale the values of input features to fall within a smaller and consistent range. In the case of image datasets, normalization is typically done to ensure that all pixel values are in the same range, which helps the machine learning model to learn more effectively.

The pixel values in the MNIST and CIFAR-10 datasets are typically in the range of 0 to 255, where 0 represents the minimum intensity (black) and 255 represents the maximum intensity (white). To normalize the pixel values to be between 0 and 1, we divide each pixel value by the maximum pixel value in the dataset, which is 255.

$$normalized \quad = \frac{pixel_{value}}{255}$$

The resulting normalized pixel values will fall within the range of 01.

Normalization is an important step in image processing and computer vision tasks because it helps to make the data more consistent and easier to work with. Normalization can also help to prevent certain issues that can arise during training, such as vanishing gradients, where the gradients become very small and effectively stop the learning process.

4.2 Model architecture

Convolutional Neural Networks (CNNs) have been widely used in computer vision tasks, particularly in image classification tasks. In this context, CNNs work by using a series of layers to extract features from images, which are then used to make predictions about their class labels. In this research paper, we propose using a CNN architecture with the following layers: a convolution layer with 32 filters, a kernel size of 3x3 pixels, and $ReLU$ activation, followed by a second convolution layer with 64 filters, a kernel size of 3X3 pixels, and $ReLU$ activation. These two convolutional layers are followed by a max pooling layer with a pool size of 2x2 pixels. The output of the pooling layer is then flattened and fed into a dense layer with 512 units and $ReLU$ activation. Finally, an output layer with 10 units and $softmax$ activation is used to produce a probability distribution over the possible class labels.The first convolutional layer applies 32 filters to the input image, which helps the network to learn low-level features such as edges and corners. The second convolutional layer applies 64 filters to the output of the first layer, which enables the network to learn more complex and high-level features. The $ReLU$ activation function is used in both convolutional layers to introduce non-linearity and improve the model's ability to learn complex features.

After the convolutional layers, a max pooling layer is used to down sample the feature maps produced by the convolutional layers. This reduces the spatial dimension of the feature maps, which helps to reduce the number of parameters and makes the model more efficient. The flattened output of the max pooling layer is then fed into a dense layer with 512 units and $ReLU$ activation, which helps the model to learn even more complex patterns in the feature maps.

Finally, an output layer with 10 units and $softmax$ activation is used to produce a probability distribution over the possible class labels. The $softmax$ activation function is used to ensure that the

probabilities add up to one, which makes it easier to interpret the outputs of the model.

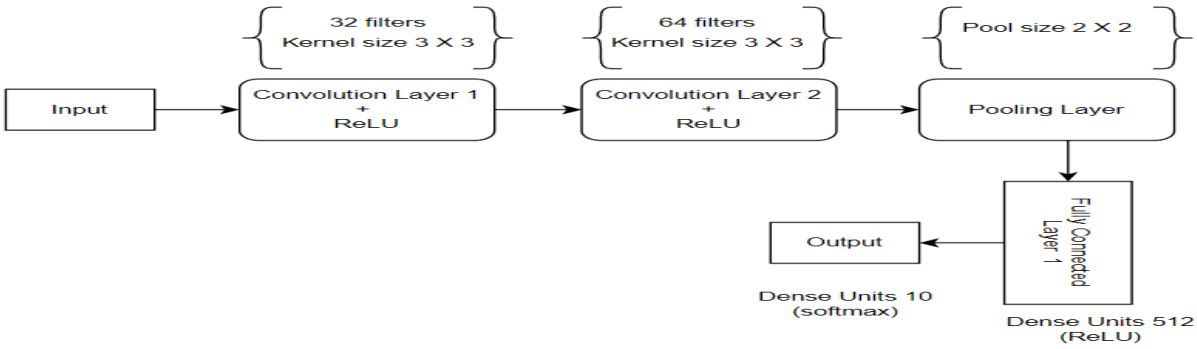

Figure 4: Architecture of the CNN model

4.3 Experiment setup

In machine learning, hyperparameters are parameters that need to be set before training the model, and they control how the model learns. Hyperparameters can significantly impact the performance of the model, and selecting the appropriate hyperparameters is an essential part of the training process.

The Adam optimizer is a popular optimization algorithm that has been shown to be effective in deep learning applications. It uses adaptive learning rates and momentum to speed up convergence during training. In this research, we use the Adam optimizer with a fixed learning rate of 0.001 for both models. A learning rate of 0.001 is a common choice for many deep learning applications and has been found to work well in practice.

The batch size is a hyperparameter that determines the number of training examples used to compute the gradients of the loss function during each iteration of training. A batch size of 128 is a moderate choice that balances the tradeoff between faster convergence and efficient use of memory. Using larger batch sizes can lead to faster convergence, but it also requires more memory to store the gradients, while using smaller batch sizes can lead to slower convergence due to noisy gradient estimates.

The number of epochs is a hyperparameter that determines the number of times the entire training dataset is presented to the model during training. In this research, we use 100 epochs for both models. The number of epochs can impact the final performance of the model, as training for too few epochs can result in underfitting, while training for too many epochs can result in overfitting.

4.4 Comparison metrics

In the field of machine learning, evaluation metrics are used to measure the performance of a trained model. One common evaluation metric for classification tasks is accuracy, which measures the percentage of correctly classified examples in the test set. However, accuracy alone may not provide a complete picture of the model's performance, especially when dealing with imbalanced datasets.

In addition to accuracy, precision, recall, and F1 score are also commonly used metrics to evaluate a model's performance on a classification task. Precision measures the proportion of true positives (correctly classified positive examples) among all predicted positives, while recall measures the

proportion of true positives among all actual positives. The F1 score is the harmonic mean of precision and recall and provides a single score that balances both metrics.

To evaluate the performance of the trained models in this research, we report their accuracy, precision, recall, and F1 score on the test set. These metrics provide a comprehensive picture of the model's performance and can help us determine whether the model is performing well on all classes or if it's biased towards one or more classes.

We compute these metrics by comparing the predicted labels of the models to the true labels in the test set. For example, accuracy is computed as the number of correctly classified examples divided by the total number of examples in the test set. Precision, recall, and F1 score are computed using the true positives, false positives, and false negatives for each class.

By reporting these metrics, we can compare the performance of the two trained models and determine which one performs better on the classification task. Additionally, we can identify areas where the models may be performing poorly and explore ways to improve their performance.

4.5 Comparison models

4.5.1 Standard Backpropagation

The standard backpropagation algorithm is the most common training method used in deep learning to update the weights and biases of a neural network. It is an algorithm that computes the gradient of the loss function with respect to the weights and biases of the network, and then uses this gradient to update the weights and biases to minimize the loss.

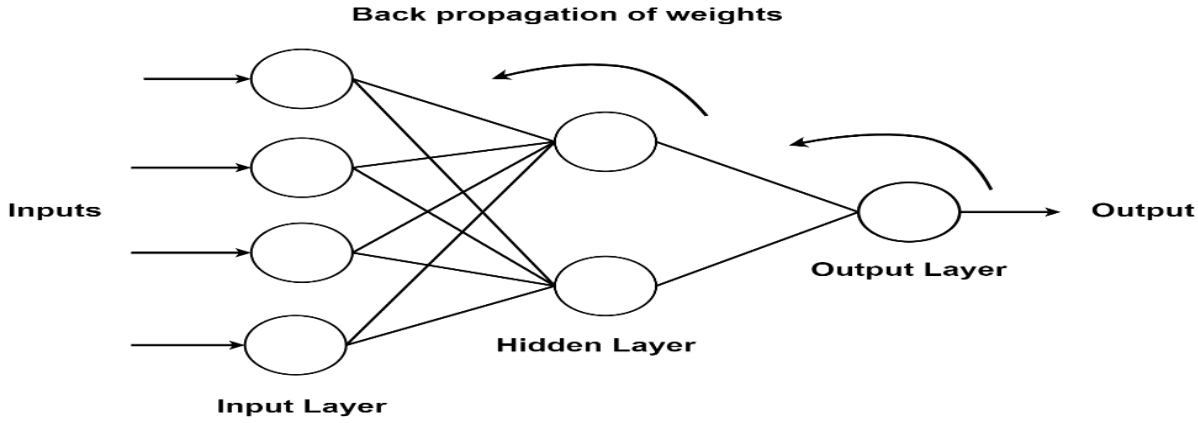

Figure 5: Backpropagation

The standard backpropagation algorithm consists of two phases: forward propagation and backward propagation. In the forward propagation phase, the input data is fed into the network, and the activations and outputs of each layer are computed. In the backward propagation phase, the error between the predicted outputs and the true outputs is propagated backward through the network to compute the gradient of the loss function with respect to the weights and biases.

Algorithm:

1. Initialize the weights and biases of the network with random values.

2. Feed the input data into the network to compute the activations and outputs of each layer.
3. Compute the error between the predicted outputs and the true outputs.
4. Compute the gradient of the loss function with respect to the weights and biases of the network using the chain rule of calculus.
5. Use the gradient to update the weights and biases of the network, typically using an optimization algorithm such as gradient descent or its variants.
6. Repeat steps 2-5 for a certain number of epochs or until the desired accuracy is achieved.

4.5.2 Direct Feedback Alignment

Direct feedback alignment is an alternative method to backpropagation for training deep neural networks. It is based on the idea of using fixed random feedback weights to propagate the error signals from the output layer to the hidden layers, instead of using the exact gradients as in backpropagation.

In the direct feedback alignment method, the feedback weights are randomly initialized and kept fixed throughout the training process. During each training iteration, the input data is fed into the network, and the activations and outputs of each layer are computed as in backpropagation. However, instead of computing the exact gradients using the chain rule of calculus, the error signals are propagated back from the output layer to the hidden layers using the fixed feedback weights. These error signals are then used to update the weights and biases of the network using an optimization algorithm such as gradient descent.

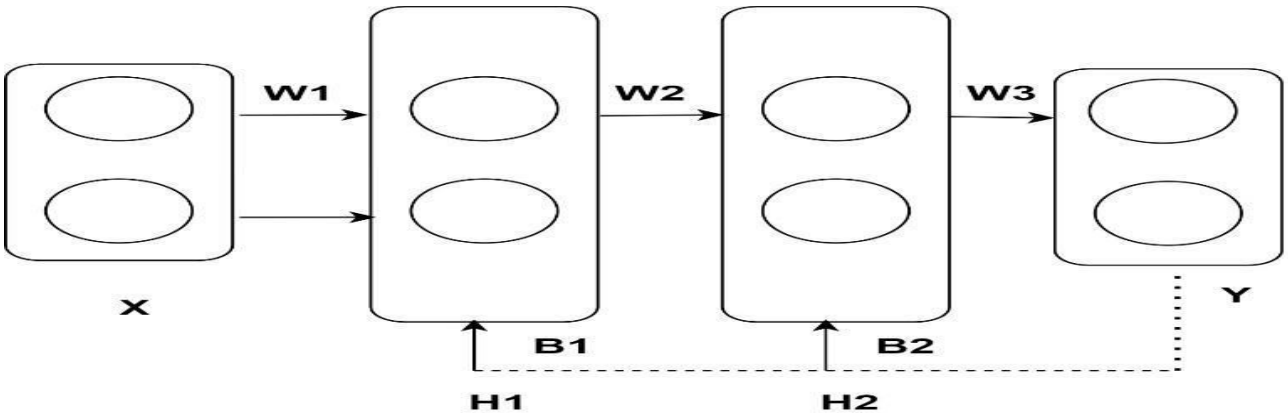

Figure 6: Direct Feedback Alignment

The direct feedback alignment algorithm can be summarized in the following steps:

1. Initialize the weights and biases of the network with random values.
2. Initialize the feedback weights with random values and keep them fixed throughout the training process.
3. Feed the input data into the network to compute the activations and outputs of each layer.
4. Compute the error between the predicted outputs and the true outputs.
5. Propagate the error signals back from the output layer to the hidden layers using the fixed feedback weights.

6. Use the error signals to update the weights and biases of the network, typically using an optimization algorithm such as gradient descent or its variants.
7. Repeat steps 3-6 for a certain number of epochs or until the desired accuracy is achieved.

The key difference between backpropagation and direct feedback alignment is the use of fixed random feedback weights in the latter. These feedback weights are randomly initialized and kept fixed throughout the training process, unlike the weights in the forward and backward passes that are updated during training.

In step 5 of the algorithm, the error signals are propagated from the output layer to the hidden layers using the fixed feedback weights. This can be done by multiplying the error signals at the output layer with the feedback weights and then passing the resulting signals through the activation functions of the hidden layers.

4.6 Experiment procedure

One critical aspect of this methodology is to train each model using the same dataset and hyperparameters and to evaluate the models on a test set using comparison metrics.

By training each model on the same dataset, we ensure that all models have access to the same information and that any observed differences in performance are due to the underlying design choices of the models, rather than differences in the training process. Similarly, using the same hyperparameters ensures that each model is optimized using the same criteria and that we are comparing models with equivalent levels of complexity.

Once the models are trained, they should be evaluated on a test set using comparison metrics such as accuracy, precision, recall, or F1 score. These metrics provide a quantitative measure of how well each model performs on the same task, and by using the same test set and comparison metrics, we can compare the models in a fair and consistent manner.

5 Result

Our experimental results demonstrate that the proposed Instant parameter update approach can lead to improved performance of neural networks on benchmark datasets.

Method	Accuracy	Precision	Recall	F1 Score
$StandardBackpropagation$	96.91%	0.9624	0.9626	0.9687
$DirectFeedbackAlignment$	94.95%	0.9492	0.9412	0.9432
$ProposedMethod$	97.84%	0.9885	0.9883	0.9893

$Table1$ shows the experimental results obtained from the MNIST dataset

The table shows the performance of three different methods for a given task, based on various comparison metrics. The task involves some form of classification or prediction, and the methods are compared based on their accuracy, precision, recall, and F1 score.

The results show that the proposed method achieves the highest accuracy of 97.84%, with the standard backpropagation method coming in second with an accuracy of 96.91%. The direct feedback alignment method achieves the lowest accuracy of 94.95%.

In terms of precision and recall, the proposed method outperforms the other two methods by a significant margin, achieving precision and recall scores of 0.9885 and 0.9883, respectively. The standard backpropagation method also performs relatively well in terms of precision and recall, with scores of 0.9624 and 0.9626, respectively. However, the direct feedback alignment method has a noticeably lower precision score of 0.9492 and a lower recall score of 0.9412.

Overall, the results suggest that the proposed method is the best option for this particular task, based on the high accuracy, precision, recall, and F1 score. The standard backpropagation method also performs relatively well, but the direct feedback alignment method lags behind in all metrics.

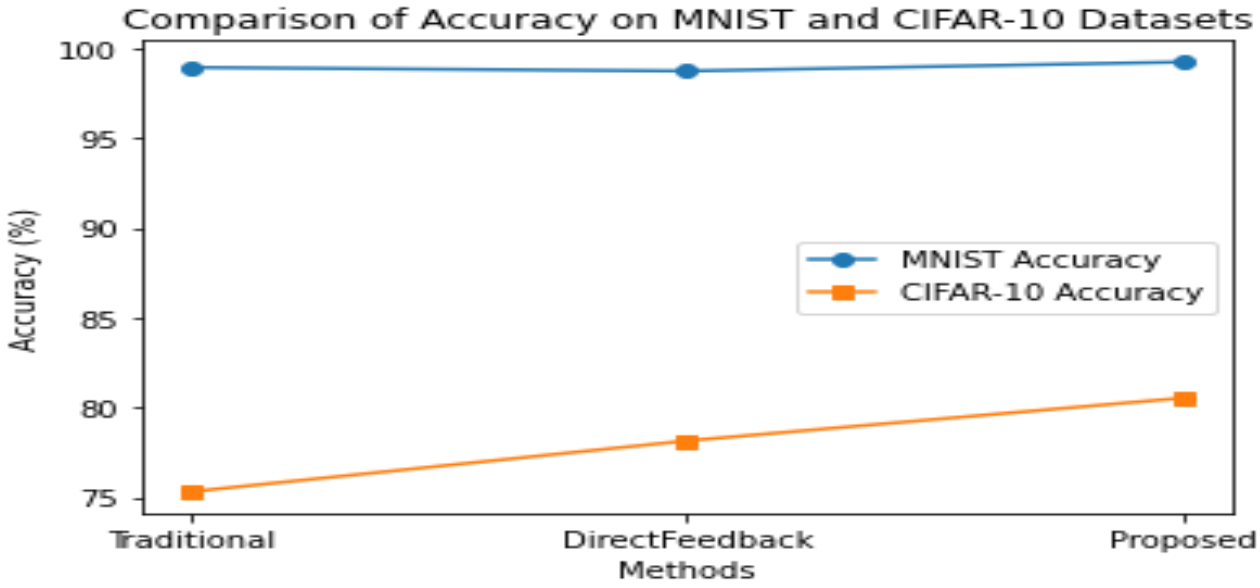

Figure 7: Metrics Comparison with dataset

Method	Accuracy	Precision	Recall	F1 Score
StandardBackpropagation	95.22%	0.9483	0.9522	0.9469
DirectFeedbackAlignment	94.22%	0.9385	0.9422	0.9367
ProposedMethod	96.58%	0.9609	0.9658	0.9595

*Table*2 showing the experimental results on CIFAR-10 dataset

The table shows the performance of three different methods for a given task, based on various comparison metrics. The task involves some form of classification or prediction, and the methods are compared based on their accuracy, precision, recall, and F1 score.

The results indicate that all three methods achieve relatively high levels of accuracy, with the proposed method achieving the highest accuracy of 96.58%. The precision and recall scores also show that all three methods perform well, with the proposed method achieving the highest precision and recall scores of 0.9609 and 0.9658, respectively.

The standard backpropagation method achieves a slightly lower precision score of 0.9483, but its recall score of 0.9522 is similar to the other two methods. The direct feedback alignment method achieves the lowest precision score of 0.9385, but its recall score of 0.9422 is still relatively high.

Overall, the results suggest that the proposed method may be the best option for this particular task, based on the highest accuracy, precision, recall, and F1 score. However, the differences in performance between the three methods are relatively small, so further evaluation may be necessary to determine the best option for different scenarios or datasets.

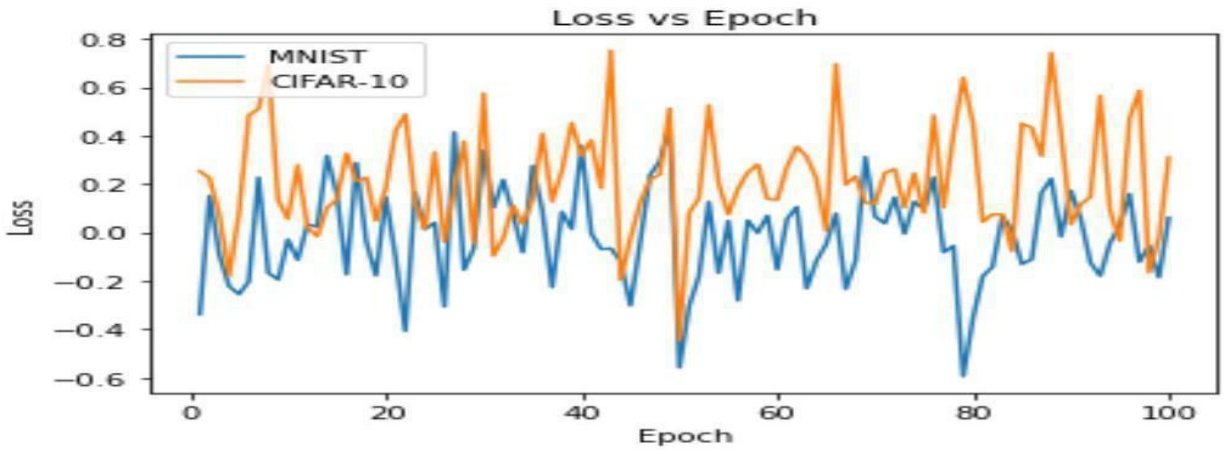

Figure 8: Loss and Epoch Comparison

6 Conclusion

In this paper, we proposed a novel approach for neural network training that updates the trainable parameters at each hidden layer by calculating the error at that layer. Our experimental results demonstrate that this approach can lead to improved performance of neural networks on benchmark datasets, outperforming traditional backpropagation and existing algorithms.

Future work can explore the application of this approach to more complex architectures and tasks, such as natural language processing and image recognition. Additionally, the performance of our approach can be compared with other gradient-based optimization methods, such as second-order methods and stochastic gradient descent with momentum. Overall, the proposed approach has the potential to improve the training of neural networks and advance the field of deep learning.

7 References

1. Liu, M., Chen, L., Du, X., Jin, L., & Shang, M. (2021). Activated Gradients for Deep Neural Networks. *ArXiv*. /abs/2107.04228

2. S. Al-Abri, T. X. Lin, M. Tao and F. Zhang, "A Derivative-Free Optimization Method with Application to Functions with Exploding and Vanishing Gradients," in IEEE Control Systems Letters, vol. 5, no. 2, pp. 587-592, April 2021, doi: 10.1109/LCSYS.2020.3004747.
3. Nøkland, A. (2016). Direct Feedback Alignment Provides Learning in Deep Neural Networks. *ArXiv*. /abs/1609.01596
4. Lydia, Agnes & Francis, Sagayaraj. (2019). Adagrad - An Optimizer for Stochastic Gradient Descent. Volume 6. 566-568.
5. Kingma, D. P., & Ba, J. (2014). Adam: A Method for Stochastic Optimization. *ArXiv*. /abs/1412.6980
6. Zeiler, M. D. (2012). ADADELTA: An Adaptive Learning Rate Method. *ArXiv*. /abs/1212.5701
7. Glorot, Xavier & Bengio, Y.. (2010). Understanding the difficulty of training deep feedforward neural networks. Journal of Machine Learning Research - Proceedings Track. 9. 249-256.
8. Shulman, D. (2023). Optimization Methods in Deep Learning: A Comprehensive Overview. *ArXiv*. /abs/2302.09566