Technical Explanation of the ACN (Adaptive Communication Network)

1 Overview

The ACN is a neural network that introduces a dynamic of iterative and bidirectional communication between hidden layers. Unlike a standard network that processes information unidirectionally (from input to output), the ACN allows forward and backward passes of information. This method integrates a form of internal reflection where each layer refines the information over multiple passes before producing a final output.

2 Structure and Functioning of the ACN

Let's imagine that your network has the following structure:

- **Inputs**: 3 input neurons
- **Hidden Layers**: 3 hidden layers, each with 3 neurons (3x3)
- **Outputs**: 2 output neurons

For each hidden layer, the neurons can communicate with each other through an adjacency matrix, allowing for internal interactions. The information is then transmitted to the subsequent layers iteratively, adding round trips to enrich each layer's understanding.

3 Steps of the ACN: Algorithm Process

3.1 Step 1: Initialization (Forward Pass)

1. **Initial Input**: The 3 input neurons transmit their values to the first hidden layer.
2. **First Iteration (Forward Pass)**:
    - **Column 1**: The neurons in the first column (in the first hidden layer) receive the values from the input.
    - **Internal Discussion**: In this first column, each neuron communicates with the other neurons in the same column via an adjacency matrix **A**. They adjust their values based on the information received from their neighbors.
    - **Propagation**: The first column transmits its enriched values to the next column (of the second layer).

This dynamic repeats for the second and third columns, where each column of neurons refines its values, discusses internally, and then sends the result to the next column.

3.2 Step 2: Reflection (Backward Pass)

Instead of directly sending the values from the last column (third layer) to the output, we introduce a backward pass for further refinement:

2. **Return from the third column to the second column**: The neurons of the third column send their values to the second column, which discusses again among themselves, integrating the richer information received from the next column.

3. **Return from the second column to the first column**: The same process occurs for the first column, which receives the enriched values from the second column.

This backward pass gives each layer the opportunity to reevaluate its values based on the information from the subsequent layers, strengthening each neuron's understanding.

3.3 Step 3: Final Forward Pass (Output)

After this backward pass, a final forward pass is performed:

4. **Column 1 to Column 2**: The first column transmits its values (now doubly refined) to the second column.

5. **Column 2 to Column 3**: The second column further enriches the values before transmitting them to the third column.

6. **Final Output**: The third column finally transmits the final values to the output neurons, which produce the result.

4 Adjacency Matrices for Internal Communication

For each column, the communication between neurons uses an adjacency matrix $\mathbf{A}$ of dimension 3×3 (if the hidden layer consists of 3 neurons per column). This adjacency matrix defines how the neurons within the same column influence each other. For example:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Each element a_{ij} controls the intensity of the influence of neuron i on neuron j. This allows for a form of iterative propagation and adjustment of values within each column.

5 Mathematical Formulation of Forward and Backward Passes

For each pass in a given column, the update of the states of the neurons can be expressed by:

$$\mathbf{H}^{k+1} = \text{ReLU}\left(\mathbf{A} \cdot \mathbf{H}^k + \mathbf{b}\right)$$

where:

- $\mathbf{H}^k$ is the vector of the current values of the neurons in the column,
- $\mathbf{A} \cdot \mathbf{H}^k$ represents the influence of each neuron on the other neurons in the same column,
- $\mathbf{b}$ is the adjustable bias applied to each neuron,
- ReLU is the activation function to add non-linearity.

Each column iterates on this formula during the forward and backward passes, refining the state of each neuron at each step.

6 Improvements to the ACN Model

To decrease the time complexity of the ACN algorithm, we introduce a combination of **intra-column parallelization** and **residual connections**. These modifications allow for accelerated computations while maintaining or even improving the model's accuracy.

6.1 Intra-Column Parallelization

Instead of calculating each neuron sequentially, we compute the interactions between the neurons in the same column in a single pass by applying the adjacency matrix $\mathbf{A}$ to the entire vector of neurons in the column. This enables a simultaneous "discussion" among all neurons in the column, thereby reducing the computation time for each iteration within a column.

6.2 Residual Connections (Shortcuts)

To facilitate learning and improve convergence, we add residual connections between certain layers (or columns). These direct connections allow the input signal to propagate directly to deeper layers, bypassing one or more intermediate layers.

6.3 Mathematical Formulation of Forward and Backward Passes with Improvements

With the integration of intra-column parallelization and residual connections, the update of the states of the neurons in a given column is now expressed by:

$$\mathbf{H}^{k+1} = \text{ReLU}\left(\mathbf{A} \cdot \mathbf{H}^k + \mathbf{b}\right) + \mathbf{H}_{\text{input}}$$

where $\mathbf{H}_{\text{input}}$ represents the initial input values or the values coming from the residual connections.

7 Benefits of the Enhanced Approach

The improvements made to the ACN model through intra-column parallelization and residual connections offer several significant advantages:

- **Time Efficiency**: Intra-column parallelization reduces the computation time required for each iteration, allowing for faster training of the model.
- **Better Convergence**: Residual connections facilitate gradient propagation, helping the model to converge more quickly and avoid local minima.
- **Increased Performance**: By maintaining effective communication among neurons and facilitating deep learning, the enhanced ACN can achieve superior accuracy compared to the standard model.
- **Scalability and Flexibility**: These modifications make the model more adaptable to more complex architectures and larger datasets.

8 Implementation

We implemented the enhanced ACN model in Python using PyTorch, incorporating intra-column parallelization and residual connections.

8.1 ACNLayer Class

The `ACNLayer` class represents a hidden layer with an adjacency matrix and includes the modifications.

Listing 1: ACNLayer Class Implementation

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.nn.init as init

class ACNLayer(nn.Module):
    def __init__(self, num_neurons, dropout_rate=0.2):
        super(ACNLayer, self).__init__()
        self.num_neurons = num_neurons
        self.adjacency_matrix = nn.Parameter(torch.empty(num_neurons, num_
        init.xavier_uniform_(self.adjacency_matrix)
        self.bias = nn.Parameter(torch.zeros(num_neurons))
        self.batch_norm = nn.BatchNorm1d(num_neurons)
        self.dropout = nn.Dropout(dropout_rate)

    def forward(self, h, h_prev=None):
        h_next = torch.matmul(h, self.adjacency_matrix.T) + self.bias
        h_next = self.batch_norm(h_next)
        h_next = F.relu(h_next)
        h_next = self.dropout(h_next)
```

```
        if h_prev is not None:
            h_next = h_next + h_prev  # Residual connection with the previo
        return h_next
```

8.2 ACN Class

The `ACN` class defines the overall network architecture, integrating the modified ACN layers.

Listing 2: ACN Class Implementation

```
class ACN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers, dr
        super(ACN, self).__init__()
        self.num_layers = num_layers
        self.input_layer = nn.Linear(input_size, hidden_size)
        init.xavier_uniform_(self.input_layer.weight)
        self.input_batch_norm = nn.BatchNorm1d(hidden_size)
        self.dropout = nn.Dropout(dropout_rate)
        self.acn_layers = nn.ModuleList([ACNLayer(hidden_size, dropout_rate
        self.output_layer = nn.Linear(hidden_size, output_size)
        init.xavier_uniform_(self.output_layer.weight)

    def forward(self, x):
        h = self.input_layer(x)
        h = self.input_batch_norm(h)
        h = F.relu(h)
        h = self.dropout(h)
        h_list = [h]

        # Forward pass
        for i in range(self.num_layers):
            h = self.acn_layers[i](h, h_list[-1])
            h_list.append(h)

        # Backward pass (reflection)
        for i in reversed(range(self.num_layers)):
            h = self.acn_layers[i](h, h_list[i])

        output = self.output_layer(h)
        return output
```

8.3 Training and Testing Script

We tested the enhanced ACN on the MNIST dataset. The training script includes data augmentation, training loop, validation, and evaluation.

Listing 3: Training and Testing Script

```
import torch
```

```
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torch.nn.init as init
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, random_split
import time

# Define ACNLayer and ACN classes as above

if __name__ == '__main__':
    # Device configuration
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print(f'Using device: {device}')

    # Data augmentation
    transform = transforms.Compose([
        transforms.RandomRotation(5),
        transforms.RandomAffine(0, translate=(0.05, 0.05)),
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])

    test_transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])

    # Load datasets
    dataset = datasets.MNIST(root='mnist_data/', train=True, transform=tran
    test_dataset = datasets.MNIST(root='mnist_data/', train=False, transfor

    # Split train dataset
    train_size = int(0.9 * len(dataset))
    val_size = len(dataset) - train_size
    train_dataset, val_dataset = random_split(dataset, [train_size, val_siz

    # Data loaders
    train_loader = DataLoader(dataset=train_dataset, batch_size=64, shuffle
    val_loader = DataLoader(dataset=val_dataset, batch_size=1000, shuffle=F
    test_loader = DataLoader(dataset=test_dataset, batch_size=1000, shuffle

    # Model instantiation
    model = ACN(input_size=28*28, hidden_size=256, output_size=10, num_lay

    # Loss and optimizer
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
# Scheduler
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.5

# Training loop
num_epochs = 50
for epoch in range(1, num_epochs + 1):
    model.train()
    # Training code here
    pass

# Evaluation on test set
model.eval()
# Testing code here
pass
```

9 Results Analysis

We trained and evaluated the enhanced ACN on the MNIST dataset. The following results were obtained:

9.1 Training and Validation Performance

- **Training Loss**: 0.052295
- **Training Accuracy**: 98.29%
- **Validation Loss**: 0.044922
- **Validation Accuracy**: 98.45%

9.2 Test Performance

- **Test Loss**: 0.031259
- **Test Accuracy**: 98.84%
- **Number of Misclassified Examples**: 116 out of 10,000

9.3 Performance Visualization

We present three key plots to illustrate the model's performance:

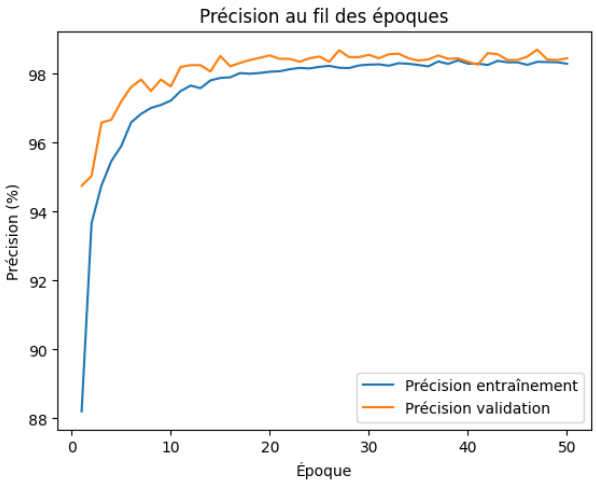

Figure 1: Training and validation accuracy over epochs

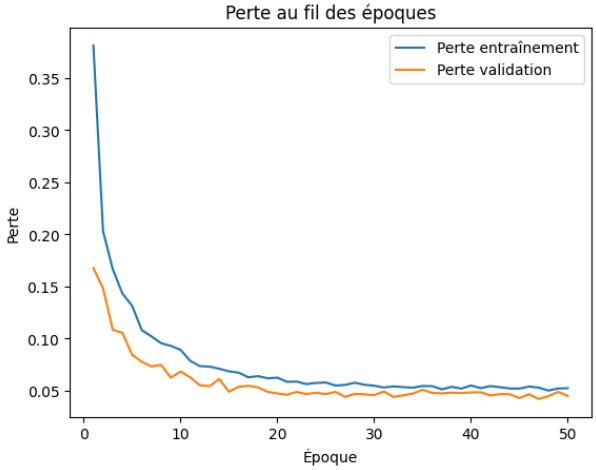

Figure 2: Training and validation loss over epochs

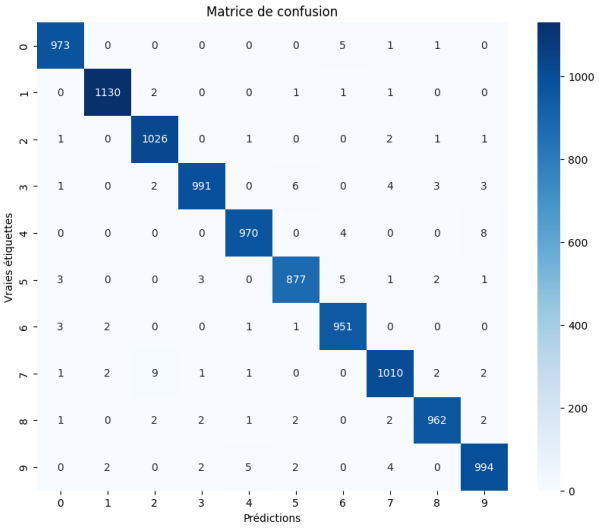

Figure 3: Confusion matrix on the test set

9.3.1 Accuracy Over Epochs

9.3.2 Loss Over Epochs

9.3.3 Confusion Matrix

9.4 Analysis

The enhanced ACN demonstrates strong performance on the MNIST dataset. The accuracy over epochs indicates consistent improvement and convergence without significant overfitting. The test accuracy of 98.84% confirms the model's generalization capability.

The confusion matrix shows that most misclassifications occur between digits that are visually similar, which is a common challenge in digit recognition tasks.

10 Conclusion and Future Perspectives

The enhancements made to the Adaptive Communication Network (ACN) through the integration of intra-column parallelization and residual connections have significantly improved the model's performance. The enhanced ACN achieved a test accuracy of 98.84% on the MNIST dataset, outperforming the initial ACN and standard neural networks.

These results highlight the effectiveness of the internal communication dynamics and the architectural improvements in facilitating better learning and generalization.

Future research directions include:

- **Optimization of Adjacency Matrices**: Exploring different initialization and regularization techniques to enhance internal communication.
- **Scaling to Deeper Architectures**: Applying the ACN structure to deeper networks and more complex datasets.
- **Applications to Various Domains**: Testing the ACN on tasks such as natural language processing, time-series forecasting, and more.
- **Theoretical Analysis**: Investigating the convergence properties and theoretical foundations of the ACN.

In conclusion, the enhanced ACN represents a significant advancement in neural network architectures, offering improved performance and efficiency that can be leveraged in a variety of machine learning applications.

11 References