Backpropagation Algorithm
Artificial Neural Networks
Project Documentation

Anton Dmitriev
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
xdmitriev@stuba.sk

Contents

1. Introduction

This project implements the backpropagation algorithm to train a neural network. The algorithm minimizes a specified loss function, enabling the network to learn effectively. To validate the implementation, a neural network is trained primarily on the XOR problem using Mean Squared Error (MSE) as the loss function.

2. Task

Implement a backpropagation algorithm for training a modular multilayer perceptron. The implementation should include support for linear layers, activation functions (Sigmoid, Tanh, ReLU), and the MSE loss function. All computations must be performed using `NumPy`.

2.1. Model

Architecture:

- Two-layer neural network:
  - 4 neurons in the hidden layer
  - 1 neuron in the output layer
- Loss function: MSE
- Supported activation functions: Sigmoid, Tanh, ReLU
- Tasks: XOR, OR, AND

Training Configuration:

- With or without momentum
- Learning rate: 0.01–0.1
- Approximately 500 epochs

Requirements:

- Implement **Forward** and **Backward** passes for all modules.
- Store intermediate results to compute gradients efficiently.
- Provide an intuitive interface:
  - `model.forward(input)` for forward propagation.
  - `model.backward(error)` for backward propagation.

3. Structure

3.1. backpropagation.py

The main script that implements the backpropagation algorithm, initializes the neural network, and manages training and evaluation. It imports configurations from `'config.py'` and contains functions for forward and backward passes and metrics visualization.

Key Functions:

- `train()`: Trains the neural network and logs training loss
- `evaluate()`: Evaluates the model's predictions on the training data
- `plot_metrics()`: Plots the training loss trend across epochs

3.2. config.py

Stores configuration parameters, including:

- **Dataset Configuration:** `dataset`, `x_train`, `y_train`
- **Neural Network Configuration:** `activation`, `support_layer`, `momentum`
- **Training Configuration:** `learning_rate`, `epochs`

4. How to Run

4.1. Run Main Script

To train and evaluate the model, run the `backpropagation.py` script:

```
python backpropagation.py
```

4.2. Adjust Parameters

Modify `'config.py'` to update:

- Dataset (`xor`, `and`, or `or`)
- Activation function (`sigmoid`, `tanh`, or `relu`)
- Use or disable momentum
- Use or disable support layer
- Learning rate and epochs

4.3. Dependencies

Install required Python libraries:

```
pip install numpy matplotlib
```

5. Architecture

5.1. Dataset

The dataset is configurable (XOR, OR, or AND) and contains input-output pairs for training logic gates. Each dataset includes:

- `x_train`: A set of binary inputs (e.g., [0, 0], [0, 1]).
- `y_train`: The expected binary outputs (e.g., [0], [1]).

```
dataset = 'xor'
datasets = {
    'xor': {
        'x_train': np.array([[0, 0], [0, 1], [1, 0], [1, 1]]),
        'y_train': np.array([[0], [1], [1], [0]])
    },
    'or': {
        'x_train': np.array([[0, 0], [0, 1], [1, 0], [1, 1]]),
        'y_train': np.array([[0], [1], [1], [1]])
    },
    'and': {
        'x_train': np.array([[0, 0], [0, 1], [1, 0], [1, 1]]),
        'y_train': np.array([[0], [0], [0], [1]])
    }
}
```

5.2. Functions

5.2.1. Activation Functions

- Activation function is mathematical function that converts input data of layer into output values that are passed to the next layer.
- Neural net supports the following activation functions:

1. **Sigmoid:**

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

2. **Tanh:**

```
def tanh(x):
    return np.tanh(x)
```

3. **ReLU:**

```
def relu(x):
    return np.maximum(0, x)
```

5.2.2. Derivatives

- Backpropagation algorithm uses derivatives to compute the gradient of the error function over the weights of the network.
- Gradient shows how the weights should be changed to reduce the error.
- Each activation function has a corresponding derivative for backpropagation:

1. **Sigmoid** Derivative:

```
def sigmoid_derivative(x):
    return x * (1 - x)
```

2. **Tanh** Derivative:

```
def tanh_derivative(x):
    return 1 - x * 2
```

3. **ReLU** Derivative:

```
def relu_derivative(x):
    return (x > 0).astype(float)
```

5.2.3. Configuration

- Functions are dynamically selected using the function:

```
def get_activation_function(name):
    if name == 'sigmoid':
        return sigmoid, sigmoid_derivative
    elif name == 'tanh':
        return tanh, tanh_derivative
    elif name == 'relu':
        return relu, relu_derivative
```

5.2.4. Loss Function

- Loss function is mathematical function that evaluates how well neural net performs its task.
- It measures the difference between predicted model values `y_pred` and actual target values `y_true`.
- The loss function used is **Mean Squared Error (MSE)**. It is used for regression problems when we need to minimize the difference between predicted and real numerical values:

```python
def mse_loss(y_true, y_pred):
    return np.mean((y_true - y_pred) * 2)

def mse_loss_derivative(y_true, y_pred):
    return 2 * (y_pred - y_true) / y_true.size
```

5.3. Layer

Layers are the building blocks of a neural network. Each layer performs a specific function: transforming input data and calculating its representation, which is then passed to the next layer.

5.3.1. Linear Layer

- **Function:** Performs a linear transformation of input data:

```
output = input * weights + bias
```

- **Weight Initialization:** Proper initialization ensures stable and efficient training, avoiding issues like vanishing or exploding gradients.
  - For **Sigmoid** and **Tanh** activations:

    ```
    weights ~ U(-limit, +limit)
    limit = sqrt(6 / (input_size + output_size))
    ```

    Note: It keeps activations within a range where gradients are large enough for effective learning, preventing saturation.
  - For **ReLU** activation:

    ```
    weights ~ N(0, std_dev^2)
    std_dev = sqrt(2 / input_size)
    ```

    Note: It ensures neurons remain active, avoiding "dead" ReLU units by maintaining balanced activations.
- **Purpose:** The linear layer computes a weighted sum of the inputs, preparing them for the non-linear transformation by the activation layer.

```python
class LinearLayer:
    def __init__(self, input_size, output_size, activation='tahn', use_momentum=False):
        if activation == 'sigmoid' or activation == 'tanh':
            limit = np.sqrt(6 / (input_size + output_size))
            self.weights = np.random.uniform(-limit, limit, (input_size, output_size))
        elif activation == 'relu':
            std_dev = np.sqrt(2 / input_size)
            self.weights = np.random.randn(input_size, output_size) * std_dev

        self.bias = np.zeros((1, output_size))
        self.learning_rate = learning_rate

        self.momentum = momentum
        self.use_momentum = use_momentum

        if self.use_momentum:
            self.weight_momentum = np.zeros_like(self.weights)
```

```
            self.bias_momentum = np.zeros_like(self.bias)

    def forward(self, x):
        ...

    def backward(self, gradient_output):
        ...
```

5.3.2. Activation Layer

- **Function:** Adds non-linearity to the model by applying a chosen activation function to the inputs. Non-linearity is crucial for the network to learn complex patterns beyond simple linear relationships.
- **Supported Activation Functions:**
  - **Sigmoid:** Maps input to the range [0, 1], useful for probabilities.
  - **Tanh:** Maps input to the range [−1, 1], useful for centered data.
  - **ReLU:** Outputs positive values as-is and zeros out negative values, commonly used in deep networks for faster convergence.
- **Purpose:** Converts the linear outputs from the `LinearLayer` class into non-linear representations, enabling the network to approximate more complex functions.

```
class ActivationLayer:
    def __init__(self, activation, activation_derivative):
        self.activation = activation
        self.activation_derivative = activation_derivative

    def forward(self, x):
        ...

    def backward(self, gradient_output):
        ...
```

5.3.3. ReLu Output Layer

When the layers use **ReLU**, Sigmoid is chosen for the output layer because, without it, the algorithm would stagnate at a constant value of **0.5**, failing to improve during training. This issue arises because the raw output values are not properly scaled, which disrupts the gradient flow and prevents the model from effectively adjusting its weights.

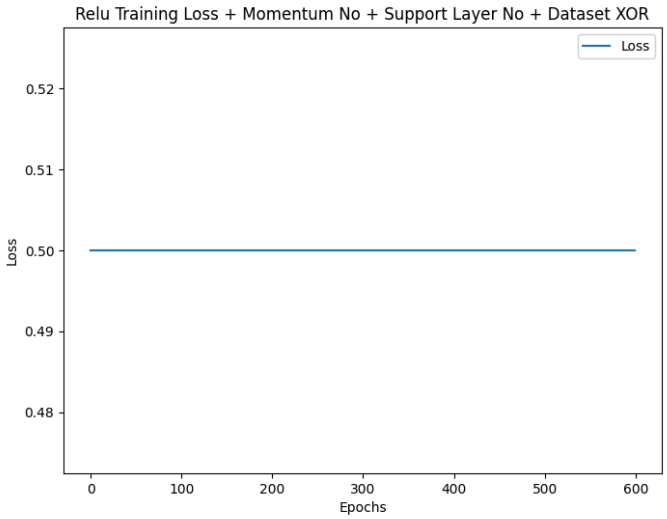

Figure 1: ReLu Stagnation

Ideally, as I think, when using ReLU, a different loss function such as **Cross-Entropy** would be more suitable for tasks requiring binary outputs, as it complements the nature of probability-like predictions. However, since the task strictly requires the use of MSE, this workaround was necessary to resolve the stagnation issue.

Using **Sigmoid** resolves this by:
- Scaling the output to the range [0, 1], ensuring the gradients remain consistent and usable.
- Preventing stagnation by enabling the algorithm to produce more meaningful outputs during training.

```
activation_func, activation_derivative = get_activation_function(activation)

if activation == 'relu':
    output_activation_func, output_activation_derivative = sigmoid, sigmoid_derivative
    output_activation_name = 'sigmoid'
else:
    output_activation_func, output_activation_derivative = activation_func,
activation_derivative
    output_activation_name = activation
```

5.4. Neural Net

The `NeuralNet` class represents the entire neural network as a sequence of layers. It combines linear and activation layers into a cohesive structure, allowing for forward and backward propagation through the network.

- **Initialization:**
  - The network is initialized with a list of layers.
  - This modular design allows flexibility in building networks of varying depths and architectures.
- **Forward Method:**
  - Iterates through the layers in order.
  - Each layer processes the input and passes its output to the next layer.
  - Example flow:

```
input -> LinearLayer1 -> ActivationLayer1 -> LinearLayer2 -> ActivationLayer2 -> output
```

- **Backward Method:**

- Iterates through the layers in reverse order (from the last to the first).
- Each layer computes its gradients and updates its parameters.
- Example flow:

```
gradient_output <- ActivationLayer2 <- LinearLayer2 <- ActivationLayer1 <- LinearLayer1
```

Purpose

- **Modularity:** Makes it easy to create networks by chaining layers in any order or configuration.
- **Forward Pass:** Transforms the input data step-by-step to produce predictions.
- **Backward Pass:** Updates the parameters of all layers using gradients computed during backpropagation.

```
class NeuralNet:
    def __init__(self, layers):
        self.layers = layers

    def forward(self, x):
        ...

    def backward(self, grad_output):
        ...
```

6. Training

The `train()` function is responsible for iteratively training the neural net by adjusting its parameters (weights and biases) to minimize the error.

6.1. Configuration

- **Learning Rate:** Set to 0.1 to prioritize faster convergence during training.
- **Number of Epochs:** Set to 600 to comply with the requirement (approximately 500 epochs) stated in the task.

6.2. Steps

1. **Initialization:**
   - `loss_history`: A list to store the loss at each epoch for tracking the network's performance.
2. **Forward Pass:**
   - The input data is passed through the network using the `model.forward(x_train)` method, producing predictions.
3. **Loss Calculation:**
   - The difference between the predictions and the true values is measured using the Mean Squared Error (MSE) loss function:

     ```
     loss = mse_loss(y_train, predictions)
     ```

   - The loss is appended to `loss_history` for further visualization and analysis.
4. **Backward Pass:**
   - The gradient of the loss with respect to the network's output is computed using `mse_loss_derivative()`.
   - The gradient is propagated back through the network using `model.backward(gradient_loss)`. This updates the weights and biases in each layer.
6. **Return:**

- The function returns `loss_history`, which can be used to plot the loss trend over epochs.

```
def train(model, epochs):
    loss_history = []

    for epoch in range(epochs):
        # forward pass
        predictions = model.forward(x_train)
        loss = mse_loss(y_train, predictions)
        loss_history.append(loss)

        # backward pass
        gradient_loss = mse_loss_derivative(y_train, predictions)
        model.backward(gradient_loss)

        if epoch % 100 == 0:
            print(f"Epoch {epoch}, Loss {loss:.4f}")

    return loss_history
```

7. Evaluating

The `evaluate()` function checks the net's predictions against the actual data to measure its performance.

7.1. Steps

1. **Forward Pass:**
   - The input data is passed through the network to produce predictions:

     ```
     predictions = model.forward(x_train)
     ```

2. **Rounding Predictions:**
   - For classification tasks, predictions are rounded to the nearest integer (e.g., for binary tasks, `prediction ≥ 0.5` are rounded to `1`; otherwise, to `0`):

     ```
     rounded_predictions = np.round(predictions).astype(int)
     ```

3. **True Values:**
   - The function prints the following:
     - Raw predictions
     - Rounded predictions (final classification output)
     - True values from the dataset

```
def evaluate(model):
    predictions = model.forward(x_train)
    rounded_predictions = np.round(predictions).astype(int)

    print("\nPredictions", predictions.flatten().tolist())
    print("Rounded", rounded_predictions.flatten().tolist())
    print("Actual", y_train.flatten().tolist())
```

8. Visualization

The `plot_metrics(loss_history, title)` function visualizes the training loss over epochs to evaluate the network's performance and convergence.

```
def plot_metrics(loss_history, title):
    plt.plot(loss_history, label="Loss")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    ...

    plt.show()
```

9. Main Function

The main function initializes and trains the neural network based on the configuration, evaluates its performance, and visualizes the results:

- **Activation Functions:** Selects the primary and output activation functions based on the configuration.
- **Model Initialization:** Builds the network by assembling layers:
  - Without a support layer: Input -> Hidden -> Output.
  - With a support layer: Input -> Hidden -> Support Hidden -> Output.
- **Training:** Calls the `train()` function to minimize the loss over the specified number of epochs.
- **Evaluation:** Uses the `evaluate()` function to compare predictions against actual values.
- **Visualization:** Plots the loss history using `plot_metrics()` to track the network's convergence.

10. Comparison

10.1. Activation Functions

10.1.1. Sigmoid

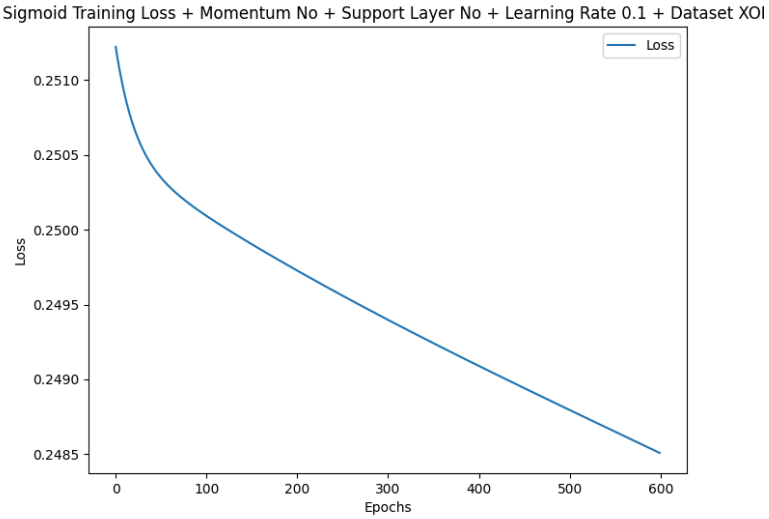

Figure 2: Sigmoid

```
...
Epoch 500, Loss 0.2488

Predictions [0.4786, 0.4616, 0.5368, 0.5104]
Rounded [0, 0, 1, 1]
Actual [0, 1, 1, 0]
```

10.1.2. Tanh

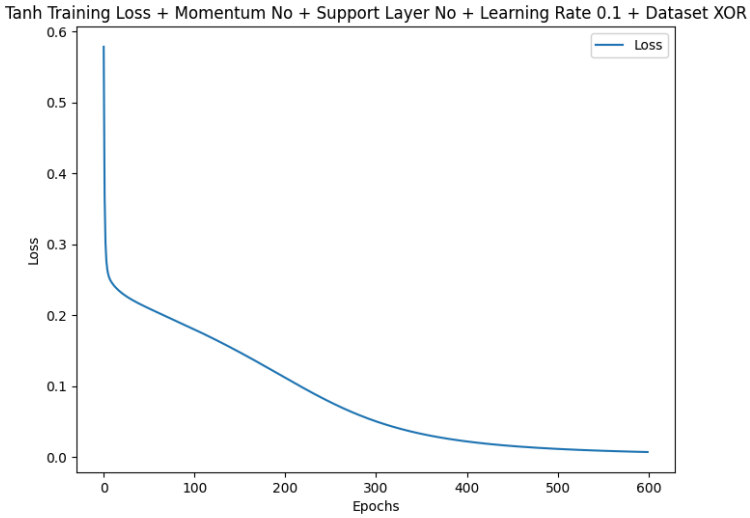

Figure 3: Tanh

```
...
Epoch 500, Loss 0.0115

Predictions [0.0174, 0.8921, 0.8774, 0.0362]
Rounded [0, 1, 1, 0]
Actual [0, 1, 1, 0]
```

10.1.3. ReLu

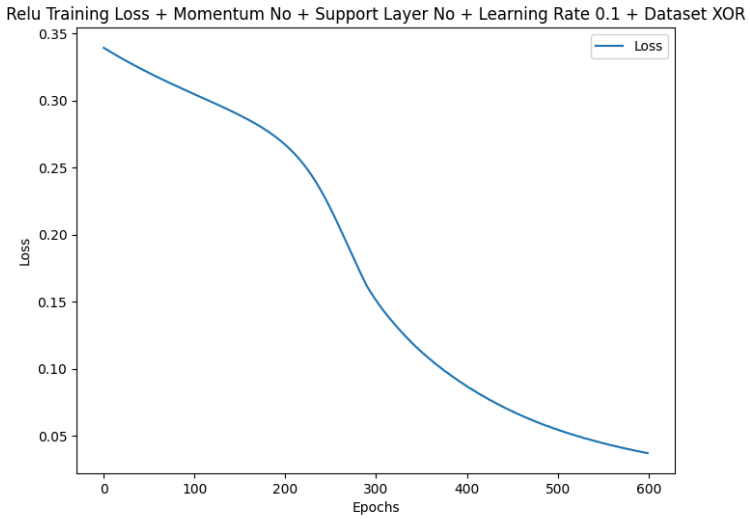

Figure 4: ReLu

```
...
Epoch 500, Loss 0.0547

Predictions [0.2625, 0.8665, 0.831, 0.1828]
Rounded [0, 1, 1, 0]
Actual [0, 1, 1, 0]
```

10.2. Momentum

- `momentum = 0.9`

10.2.1. Sigmoid + Momentum

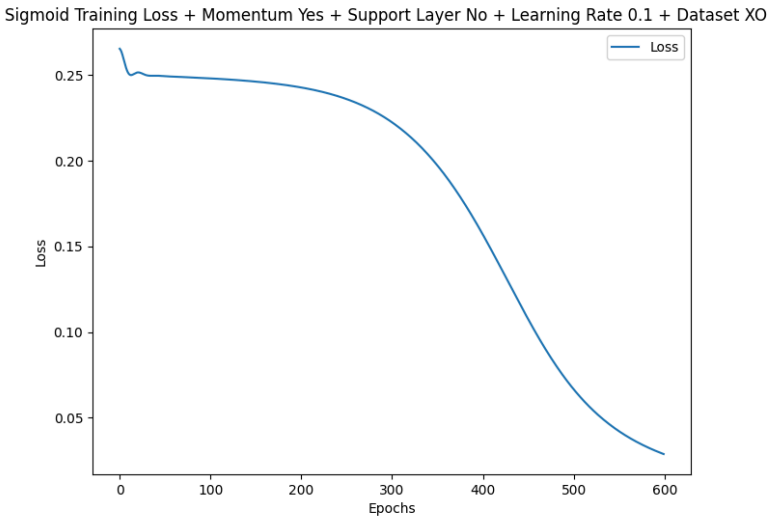

Figure 5: Sigmoid + Momentum

```
...
Epoch 500, Loss 0.0666

Predictions [0.1796, 0.8381, 0.8169, 0.1505]
Rounded [0, 1, 1, 0]
Actual [0, 1, 1, 0]
```

10.2.2. Tanh + Momentum

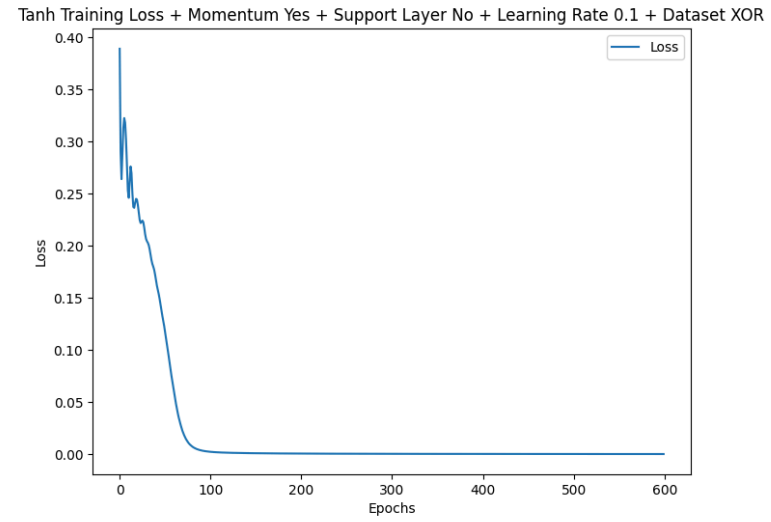

Figure 6: Tanh + Momentum

```
...
Epoch 500, Loss 0.0002

Predictions [0.0005, 0.9831, 0.9831, 0.0008]
```

```
Rounded [0, 1, 1, 0]
Actual [0, 1, 1, 0]
```

10.2.3. ReLu + Momentum

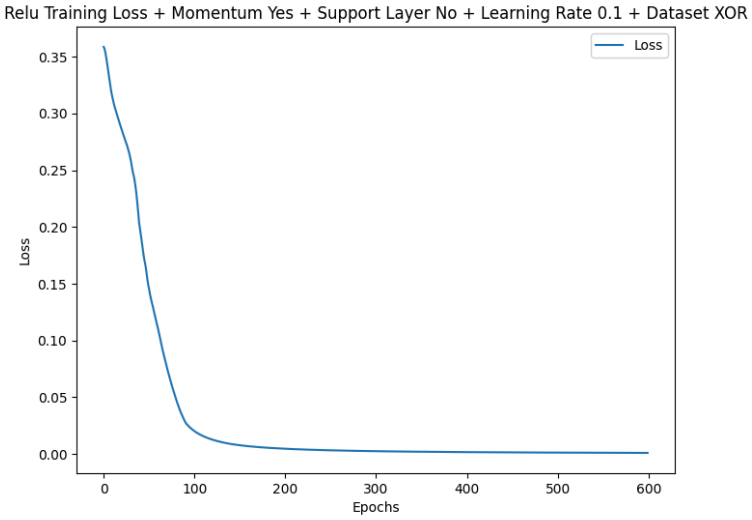

Figure 7: ReLu + Momentum

```
...
Epoch 500, Loss 0.0013

Predictions [0.0531, 0.9816, 0.9768, 0.0172]
Rounded [0, 1, 1, 0]
Actual [0, 1, 1, 0]
```

10.3. Hidden Layers

- `support_layer_in = 4`
- `support_layer_out = 4`

10.3.1. Sigmoid + Support Layer

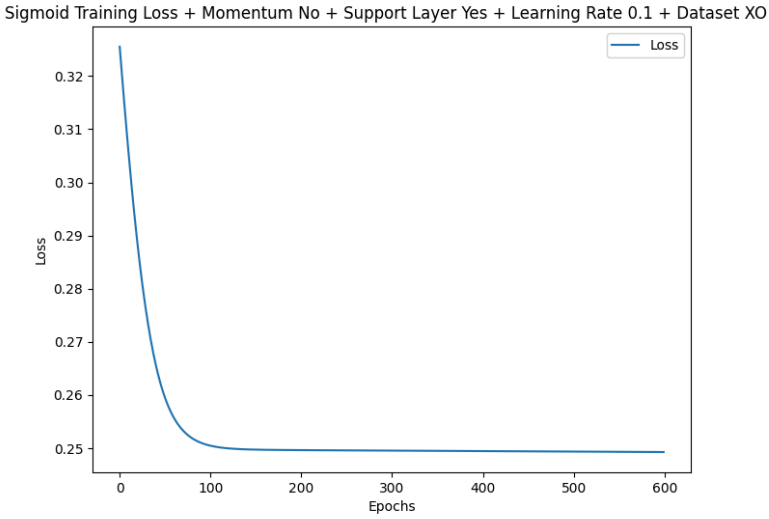

Figure 8: Sigmoid + Support Layer

```
...
Epoch 500, Loss 0.2494

Predictions [0.4937, 0.5096, 0.4927, 0.5056]
Rounded [0, 1, 0, 1]
Actual [0, 1, 1, 0]
```

10.3.2. Tanh + Support Layer

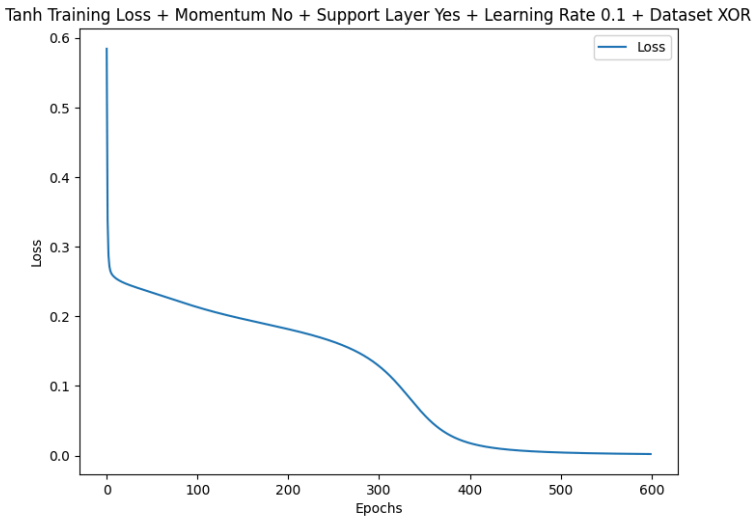

Figure 9: Tanh + Support Layer

```
...
Epoch 500, Loss 0.0044

Predictions [0.006, 0.9333, 0.9371, 0.0102]
Rounded [0, 1, 1, 0]
Actual [0, 1, 1, 0]
```

10.3.3. ReLu + Support Layer

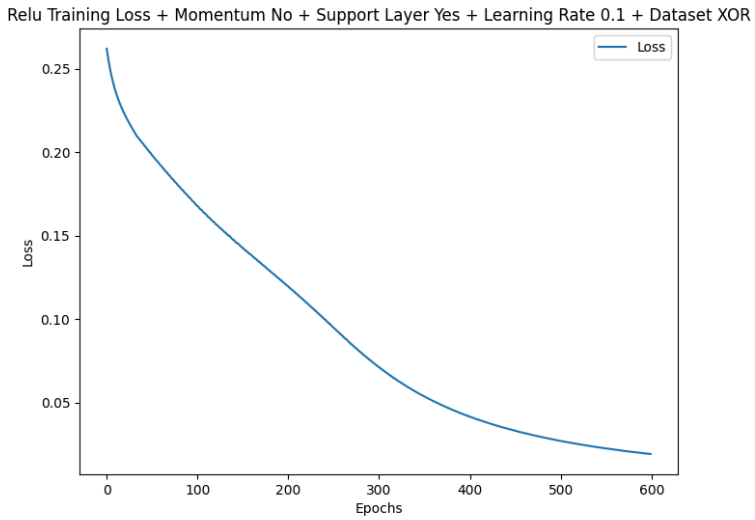

Figure 10: ReLu + Support Layer

```
...
Epoch 500, Loss 0.0269

Predictions [0.2422, 0.9008, 0.9469, 0.0695]
Rounded [0, 1, 1, 0]
Actual [0, 1, 1, 0]
```

10.4. Learning Rates

- `learning_rate = 0.01`

10.4.1. Sigmoid + Learning Rate

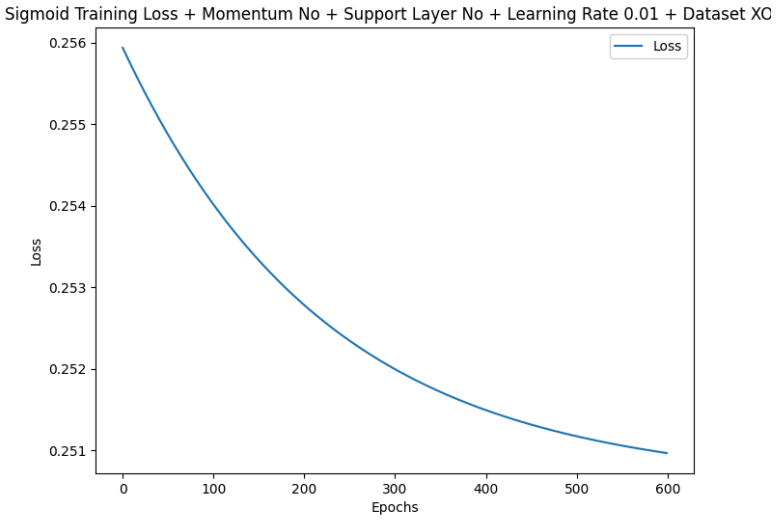

Figure 11: Sigmoid + Learning Rate

```
...
Epoch 500, Loss 0.2512

Predictions [0.4642, 0.5197, 0.4455, 0.5002]
Rounded [0, 1, 0, 1]
Actual [0, 1, 1, 0]
```

10.4.2. Tanh + Learning Rate

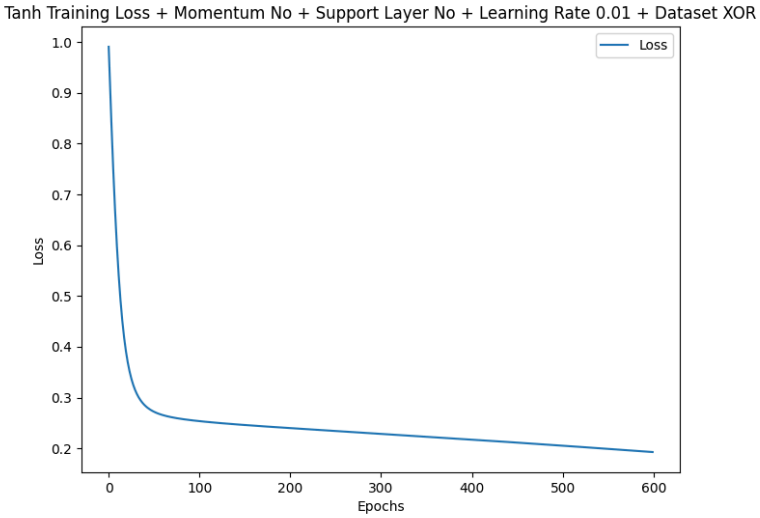

Figure 12: Tanh + Learning Rate

```
...
Epoch 500, Loss 0.2051

Predictions [0.3859, 0.5702, 0.5019, 0.434]
Rounded [0, 1, 1, 0]
Actual [0, 1, 1, 0]
```

10.4.3. ReLu + Learning Rate

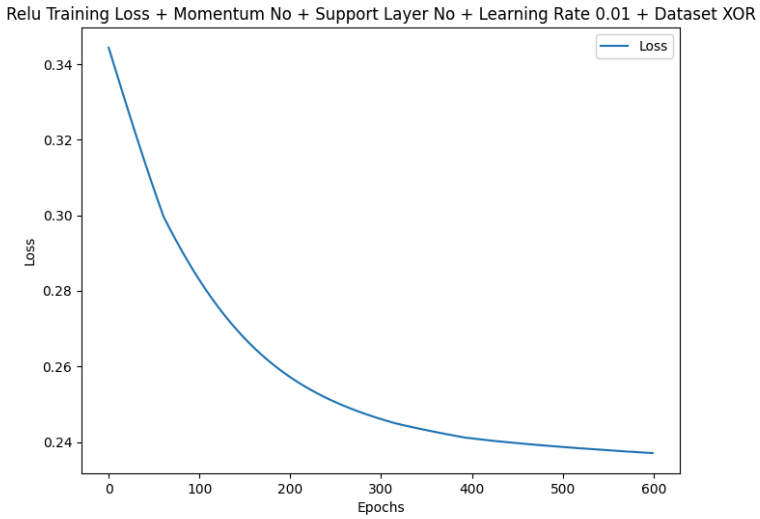

Figure 13: ReLu + Learning Rate

```
...
Epoch 500, Loss 0.2139

Predictions [0.4876, 0.5466, 0.646, 0.4952]
Rounded [0, 1, 1, 0]
Actual [0, 1, 1, 0]
```

10.5. Summary

Activation Function	Hidden Layers	Momentum	Learning Rate	Final Loss	Success
Sigmoid	1	No	0.1	0.2488	No
Sigmoid	1	Yes	0.1	0.0666	Yes
Sigmoid	2	No	0.1	0.2494	No
Sigmoid	1	No	0.01	0.2512	No
Tanh	1	No	0.1	0.0115	Yes
Tanh	1	Yes	0.1	0.0002	Yes
Tanh	2	No	0.1	0.0044	Yes
Tanh	1	No	0.01	0.2051	Yes
ReLu	1	No	0.1	0.0547	Yes
ReLu	1	Yes	0.1	0.0013	Yes
ReLu	2	No	0.1	0.0269	Yes
ReLu	1	No	0.01	0.2139	Yes

Table 1: Global Configuration Results

Observation:

1. **Momentum:** Adding momentum improves performance across all activation functions, especially for Sigmoid, which was only able to get the right result by adding momentum.
2. **Layer Depth:** Adding an extra hidden layer provides only slight improvements and is less impactful than using momentum.
3. **Learning Rate:** Lower learning rates (0.01) slow down convergence and may result in suboptimal final losses.
4. **Tanh Performance:** Tanh performs consistently well even without momentum, but adding momentum gives best results.
5. **Sigmoid Limitations:** Sigmoid struggles with XOR tasks, as I think, due to vanishing gradients and its uncentered range [0, 1], producing soft outputs unsuitable for sharp classification. Momentum helps but doesn't fully resolve these issues.

Next Steps:

- Further testing with different datasets (XOR, AND and OR) will be conducted using the best configuration:
  - **Activation Function:** Tanh
  - **Hidden Layers:** 1
  - **Momentum:** Enabled
  - **Learning Rate:** 0.1
- This configuration showed the lowest final loss (0.0002) and consistent success across tests.

10.6. Datasets

10.6.1. XOR

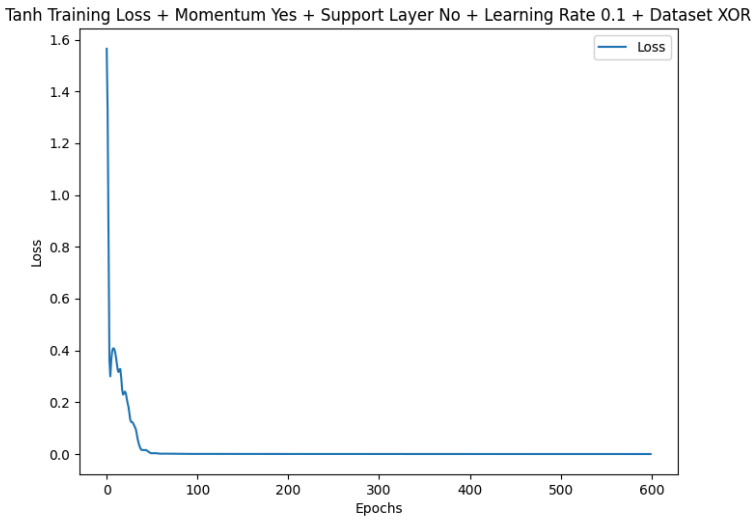

Figure 14: XOR

```
...
Epoch 500, Loss 0.0002

Predictions [0.0003, 0.9823, 0.9827, 0.0003]
Rounded [0, 1, 1, 0]
Actual [0, 1, 1, 0]
```

10.6.2. AND

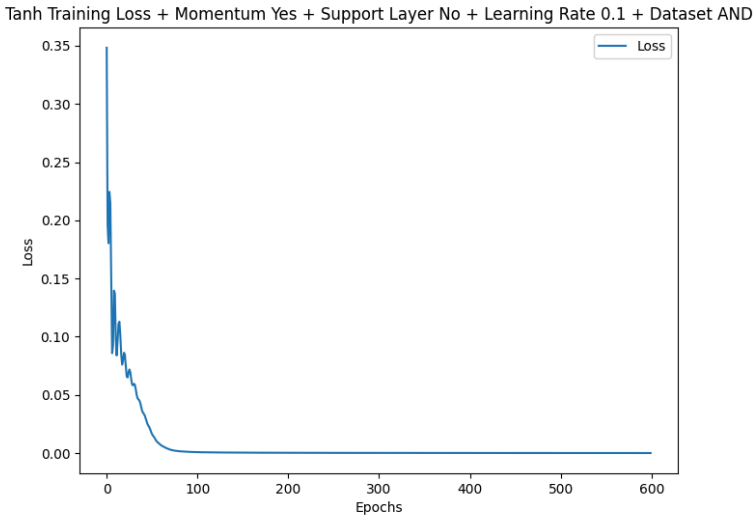

Figure 15: AND

```
...
Epoch 500, Loss 0.0001

Predictions [-0.0002, 0.0005, 0.0005, 0.9823]
```

```
Rounded [0, 0, 0, 1]
Actual [0, 0, 0, 1]
```

10.6.3. OR

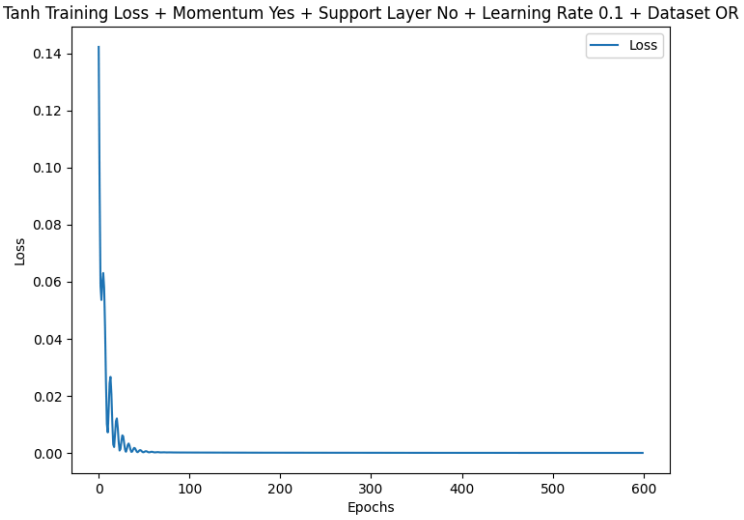

Figure 16: OR

```
...
Epoch 500, Loss 0.0000

Predictions [0.0002, 0.9909, 0.9913, 0.999]
Rounded [0, 1, 1, 1]
Actual [0, 1, 1, 1]
```

10.6.4. XOR + Support Layer

- `support_layer_in = 4`
- `support_layer_out = 4`

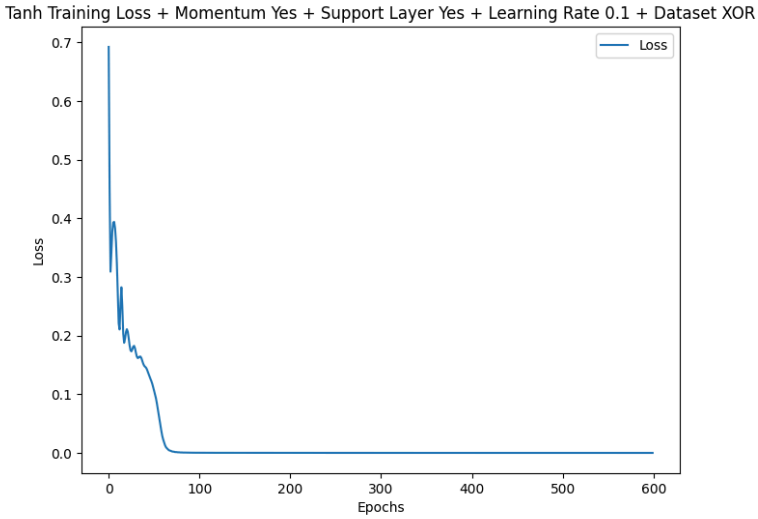

Figure 17: XOR + Support Layer

```
...
Epoch 500, Loss 0.0000

Predictions [0.0001, 0.9926, 0.9902, 0.0001]
Rounded [0, 1, 1, 0]
Actual [0, 1, 1, 0]
```

10.6.5. AND + Support Layer

- `support_layer_in = 4`
- `support_layer_out = 4`

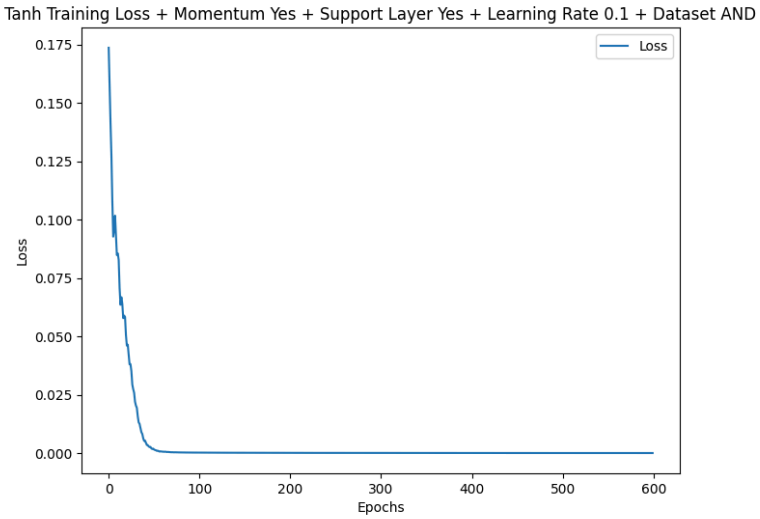

Figure 18: AND + Support Layer

```
...
Epoch 500, Loss 0.0000

Predictions [0.0001, 0.0001, 0.0002, 0.9873]
Rounded [0, 0, 0, 1]
Actual [0, 0, 0, 1]
```

10.6.6. OR + Support Layer

- `support_layer_in = 4`
- `support_layer_out = 4`

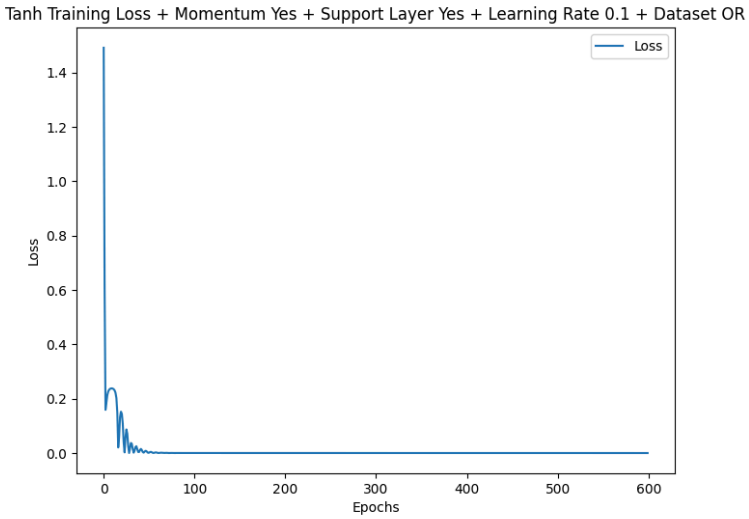

Figure 19: OR + Support Layer

```
...
Epoch 500, Loss 0.0000

Predictions [0.0, 0.9952, 0.9963, 0.9994]
Rounded [0, 1, 1, 1]
Actual [0, 1, 1, 1]
```

10.6.7. Summary

Dataset	Hidden Layers	Final Loss	Success
XOR	1	0.0002	Yes
XOR	2	0.0000	Yes
AND	1	0.0001	Yes
AND	2	0.0000	Yes
OR	1	0.0000	Yes
OR	2	0.0000	Yes

Table 2: Dataset Configuration Results

Observation:

1. **XOR, AND and OR Tasks:** All datasets achieve perfect performance (Final Loss = 0.0000) with the best configuration.
2. **Support Layers:** Adding support layers slightly improves stability and also has impact on the already excellent results.
3. **General Robustness:** The network consistently handles both linearly separable (AND, OR) and non-linear (XOR) datasets, demonstrating effective generalization.

11. Conclusion

The project successfully implements the backpropagation algorithm as specified. The neural network demonstrates strong performance across tasks, including linearly separable (AND, OR) and non-linear (XOR) datasets. Key outcomes:

- **Optimal Configuration:**
  - **Activation Function:** Tanh

- **Hidden Layers:** 2
- **Momentum:** Enabled
- **Learning Rate:** 0.1

- **Insights:**
  - Momentum improves convergence, particularly for Sigmoid.
  - Tanh achieves the best results with consistent stability.
  - Support layers offer minimal additional benefits.

These results highlight the effectiveness of the implemented architecture for small classification tasks.