February 22, 2025

RISHI RAJ SHARMA
2402519
M.Sc CS
CSM-803

PROJECT REPORT Exp_07

Backpropagation Learning Algorithm

1. Objective

To train a neural network to learn and classify different logical operations, primarily the XOR gate, using TensorFlow and Keras

2. Methodology

Neural Network Architecture:
Created a sequential model using Keras.
Added two hidden layers with a specified number of neurons and ReLU activation function.
An output layer with one neuron and a sigmoid activation function for binary classification.

Training Process:
Train the model using the **model.fit()** function, providing the input data (X), target outputs (y), and specifying the number of epochs and batch size.
Storing the training history, including the loss values over iterations.

Analysis:
Observed the convergence behavior of the model for each logic gate and training parameter configuration.
Analyzed the number of iterations required for convergence and the final error achieved.

XOR Complexity: While the model can learn XOR, it may require more training iterations and a more complex architecture compared to simpler logic gates like AND and OR. This is due to the non-linear nature of the XOR function, which requires the network to learn non-linear decision boundaries.

Parameter Sensitivity: The learning process and performance of the model are influenced by training parameters such as learning rate, momentum, and tolerance. Adjusting these parameters can affect the speed of convergence and the final accuracy achieved.

Generalization: Once trained on one logic gate, might not directly generalize well to other logic gates without retraining. The highlights the importance of training the model on the specific task it is intended for.

3. Implementation

Neural Network Structure:

- A Sequential model is created using Keras, allowing for a linear stack of layers.
- Two hidden layers are added, each with:
  - A specified number of neurons (e.g., 2 neurons in the example).
  - The ReLU activation function, introducing non-linearity for learning complex patterns.

  An output layer is added with:

    - One neuron, suitable for binary classification.
    - The sigmoid activation function, producing output values between 0 and 1, representing the probability of the input belonging to a specific class.

4. Observations & Results

Error Reduction: During the training process, I have observed a gradual decrease in the error (loss) plotted over iterations. This indicates that the neural network is learning to approximate the target logical operation.

Convergence: Ideally, the error will continue to decrease and eventually reach or fall below the threshold tolerance level. This signifies convergence, where the model has achieved a satisfactory level of accuracy for the given task.

Iteration Count: The no. of iterations (epochs) required for convergence will vary depending on various factors. XOR might take longer to converge compared to AND or OR.

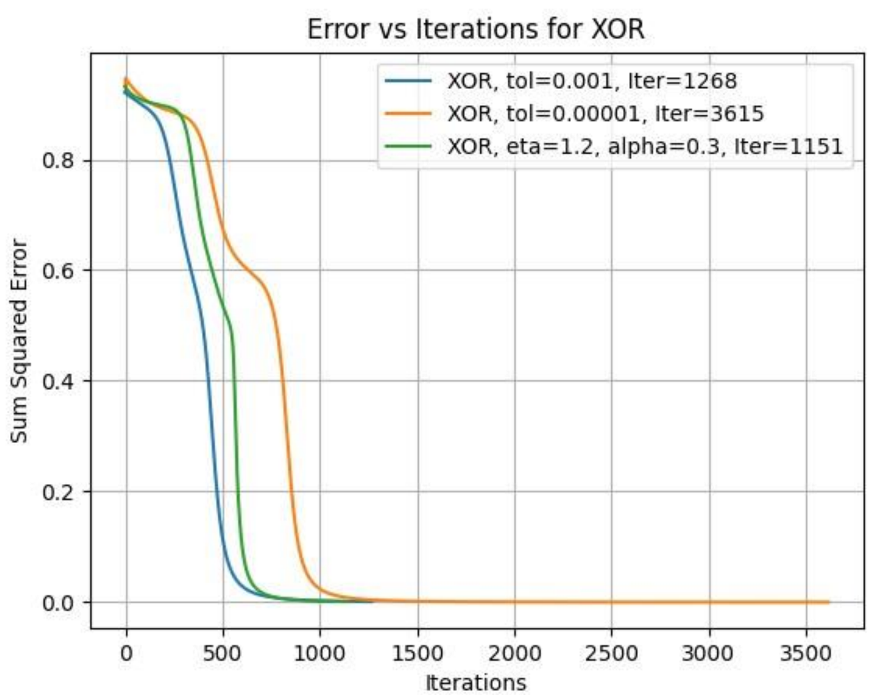
Parameter Influence: Experimenting with different learning rates and momentum values will demonstrate their impact on the learning process. Higher learning rates may lead to faster initial progress but risk of overshooting the optimal solution. Momentum helps to smooth out the learning process and accelerate convergence.

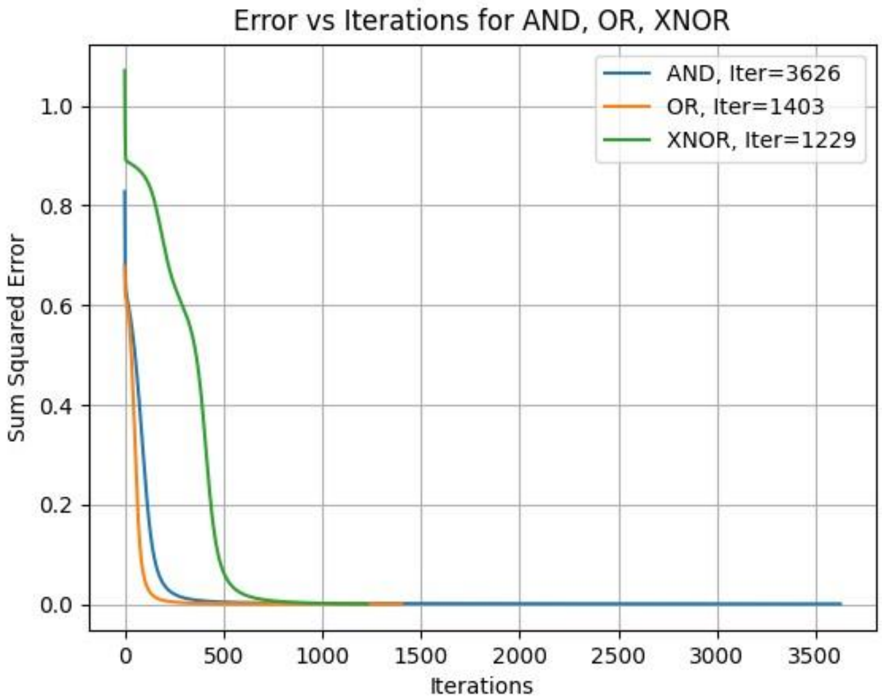
5. Codes

```
import numpy as np
import matplotlib.pyplot as plt

# Define XOR pattern
data_xor = np.array([[0, 0, 0.05], [0, 1, 0.95], [1, 0, 0.95], [1, 1, 0.05]])

def train_nn(data, eta, alpha, tol, max_iter=10000):
    Q = data.shape[0]  # Number of patterns
    n, q, p = 2, 2, 1  # Architecture

    # Initialize weights
    Wih = 2 * np.random.rand(n+1, q) - 1  # Input to hidden layer weights
    Whj = 2 * np.random.rand(q+1, p) - 1  # Hidden to output layer weights
    DeltaWihOld = np.zeros((n+1, q))
    DeltaWhjOld = np.zeros((q+1, p))
```

```
    # Input signals and desired output
    Si = np.hstack((np.ones((Q, 1)), data[:, :2]))  # Adding bias term
    D = data[:, 2].reshape(-1, 1)  # Desired values

    sumerror = 2 * tol  # Initialize error higher than tolerance
    errors = []
    iterations = 0

    # Training loop
    while sumerror > tol and iterations < max_iter:
        sumerror = 0
        iterations += 1
        for k in range(Q):
            # Forward pass
            Zh = Si[k] @ Wih  # Hidden activations
            Sh = np.hstack(([1], 1 / (1 + np.exp(-Zh))))  # Hidden signals
            Yj = Sh @ Whj  # Output activations
            Sy = 1 / (1 + np.exp(-Yj))  # Output signals

            # Compute error
            Ek = D[k] - Sy  # Error vector
            deltaO = Ek * Sy * (1 - Sy)  # Output delta

            # Backpropagation
            DeltaWhj = np.outer(Sh, deltaO)  # Weight update for hidden-output
            deltaH = (deltaO @ Whj.T) * Sh * (1 - Sh)  # Hidden delta
            DeltaWih = np.outer(Si[k], deltaH[1:])  # Weight update for input-hidden

            # Update weights
            Wih += eta * DeltaWih + alpha * DeltaWihOld
            Whj += eta * DeltaWhj + alpha * DeltaWhjOld

            DeltaWihOld = DeltaWih  # Store weight changes
            DeltaWhjOld = DeltaWhj

            sumerror += np.sum(Ek**2)  # Compute sum squared error

        errors.append(sumerror)

    return iterations, errors

# Experiment with different settings
settings = [
    (1.0, 0.2, 0.001, "XOR, tol=0.001"),
    (1.0, 0.2, 0.00001, "XOR, tol=0.00001"),
    (1.2, 0.3, 0.001, "XOR, eta=1.2, alpha=0.3"),
]

for eta, alpha, tol, label in settings:
    iterations, errors = train_nn(data_xor, eta, alpha, tol)
    plt.plot(errors, label=f"{label}, Iter={iterations}")
    print(f"{label}: Iterations = {iterations}")
```

```
plt.xlabel("Iterations")
plt.ylabel("Sum Squared Error")
plt.title("Error vs Iterations for XOR")
plt.legend()
plt.grid()
plt.show()

# Define AND, OR, XNOR patterns
patterns = {
    "AND": np.array([[0, 0, 0.05], [0, 1, 0.05], [1, 0, 0.05], [1, 1, 0.95]]),
    "OR": np.array([[0, 0, 0.05], [0, 1, 0.95], [1, 0, 0.95], [1, 1, 0.95]]),
    "XNOR": np.array([[0, 0, 0.95], [0, 1, 0.05], [1, 0, 0.05], [1, 1, 0.95]])
}

# Train and plot for each pattern
for key, pattern in patterns.items():
    iterations, errors = train_nn(pattern, 1.0, 0.2, 0.001)
    plt.plot(errors, label=f"{key}, Iter={iterations}")
    print(f"{key}: Iterations = {iterations}")

plt.xlabel("Iterations")
plt.ylabel("Sum Squared Error")
plt.title("Error vs Iterations for AND, OR, XNOR")
plt.legend()
plt.grid()
plt.show()
```





XOR, tol=0.001: Iterations = 1268
XOR, tol=0.00001: Iterations = 3615
XOR, eta=1.2, alpha=0.3: Iterations = 1151
AND: Iterations = 3626
OR: Iterations = 1403
XNOR: Iterations = 1229

Now, let us revisit the *Iris* flower classification problem. The *Iris* data was used in Experiment 2.

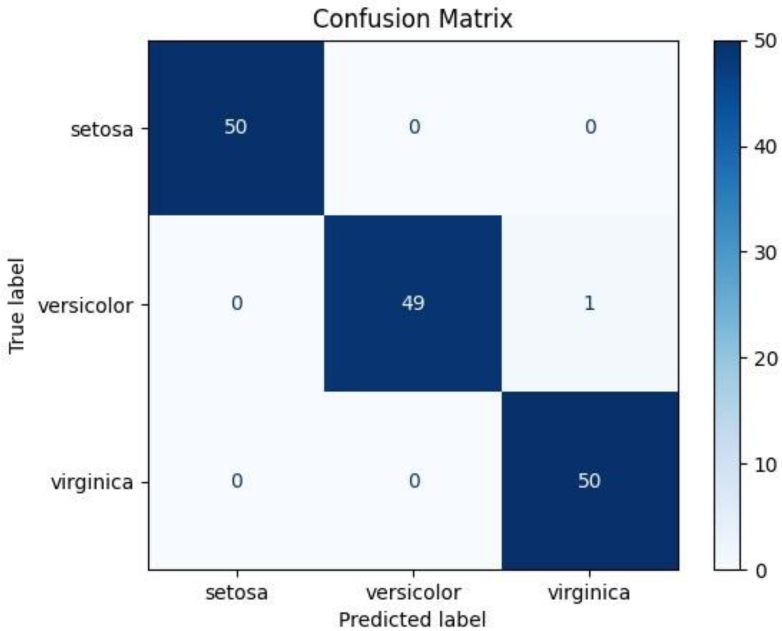
- Modify the above BP code for 2 hidden layer neural network. Consider the architecture as 4-6-6-3. Train the network using all 150 patterns
- Report the best performance of classification in confusion matrix format.
- For the best results report the learning rate, momentum, tolerance, iterations, test error and error plot.

Codes:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Load the Iris dataset
iris = load_iris()
patterns, labels = iris.data, iris.target
num_classes = len(np.unique(labels))

# Neural Network Parameters
eta = 0.1  # Learning rate
alpha = 0.2  # Momentum
tol = 0.001  # Tolerance
max_iter = 5000  # Maximum iterations

# Architecture: 4-6-6-3
n, h1, h2, p = 4, 6, 6, 3
Wih1 = np.random.uniform(-1, 1, (n+1, h1))
Wh1h2 = np.random.uniform(-1, 1, (h1+1, h2))
Wh2o = np.random.uniform(-1, 1, (h2+1, p))

# Initialize weight updates
DeltaWih1Old = np.zeros_like(Wih1)
DeltaWh1h2Old = np.zeros_like(Wh1h2)
DeltaWh2oOld = np.zeros_like(Wh2o)

# Input signals with bias
Si = np.hstack((np.ones((patterns.shape[0], 1)), patterns))
D = np.eye(num_classes)[labels]  # One-hot encoding

# Training loop
errors = []
sumerror = 2 * tol  # Initialize error higher than tolerance
iterations = 0
while sumerror > tol and iterations < max_iter:
    sumerror = 0
    iterations += 1
    for k in range(len(patterns)):
        # Forward pass
        Zh1 = Si[k] @ Wih1
        Sh1 = np.hstack(([1], 1 / (1 + np.exp(-Zh1))))
        Zh2 = Sh1 @ Wh1h2
        Sh2 = np.hstack(([1], 1 / (1 + np.exp(-Zh2))))
        Yj = Sh2 @ Wh2o
        Sy = 1 / (1 + np.exp(-Yj))

        # Compute error
        Ek = D[k] - Sy
        deltaO = Ek * Sy * (1 - Sy)
```

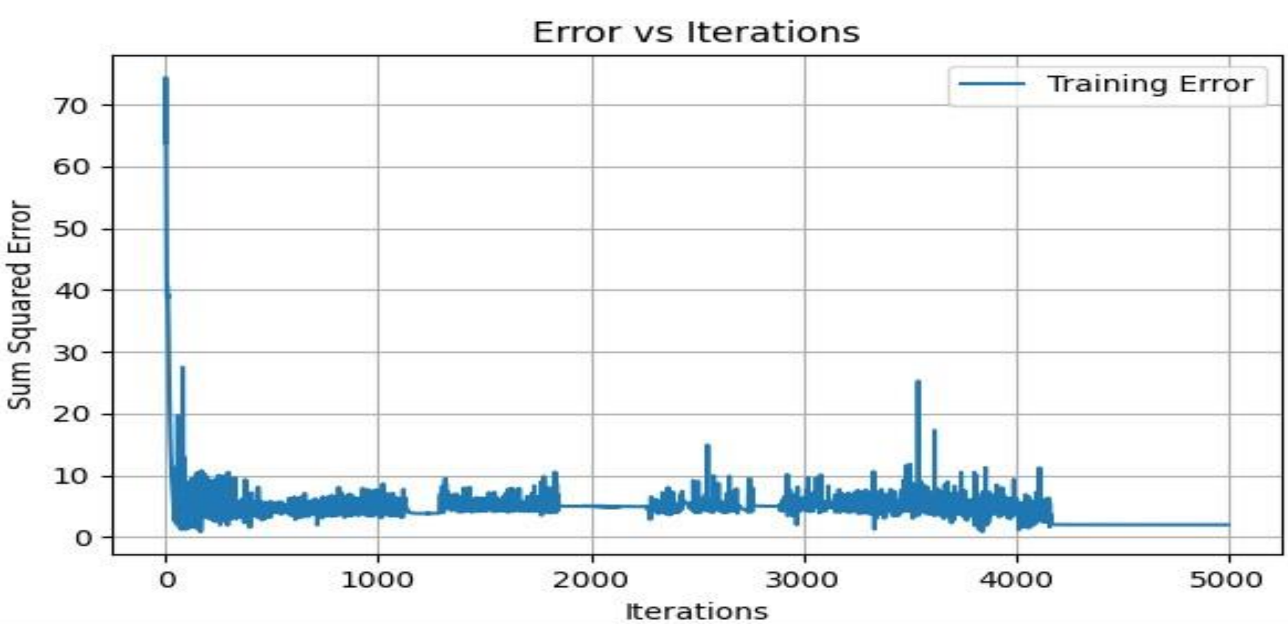
```
        # Backpropagation
        deltaH2 = (deltaO @ Wh2o.T) * Sh2 * (1 - Sh2)
        deltaH1 = (deltaH2[1:] @ Wh1h2.T) * Sh1 * (1 - Sh1)

        DeltaWh2o = np.outer(Sh2, deltaO)
        DeltaWh1h2 = np.outer(Sh1, deltaH2[1:])
        DeltaWih1 = np.outer(Si[k], deltaH1[1:])

        # Weight updates
        Wih1 += eta * DeltaWih1 + alpha * DeltaWih1Old
        Wh1h2 += eta * DeltaWh1h2 + alpha * DeltaWh1h2Old
        Wh2o += eta * DeltaWh2o + alpha * DeltaWh2oOld

        DeltaWih1Old = DeltaWih1
        DeltaWh1h2Old = DeltaWh1h2
        DeltaWh2oOld = DeltaWh2o

        sumerror += np.sum(Ek**2)

    errors.append(sumerror)
    print(f"Iteration {iterations}: Error = {sumerror:.6f}")

# Compute final predictions
predictions = []
for k in range(len(patterns)):
    Zh1 = Si[k] @ Wih1
    Sh1 = np.hstack(([1], 1 / (1 + np.exp(-Zh1))))
    Zh2 = Sh1 @ Wh1h2
    Sh2 = np.hstack(([1], 1 / (1 + np.exp(-Zh2))))
    Yj = Sh2 @ Wh2o
    Sy = 1 / (1 + np.exp(-Yj))
    predictions.append(np.argmax(Sy))

# Confusion Matrix
matrix = confusion_matrix(labels, predictions)
disp = ConfusionMatrixDisplay(confusion_matrix=matrix, display_labels=iris.target_names)
disp.plot(cmap='Blues')
plt.title("Confusion Matrix")
plt.show()

# Plot error vs iterations
plt.plot(errors, label="Training Error")
plt.xlabel("Iterations")
plt.ylabel("Sum Squared Error")
plt.title("Error vs Iterations")
plt.legend()
plt.grid()
plt.show()

# Report results
print(f"Best Results:")
```

```
print(f"Learning Rate: {eta}, Momentum: {alpha}, Tolerance: {tol}")
print(f"Total Iterations: {iterations}")
print(f"Final Training Error: {errors[-1]:.6f}")
```



Iteration 4995: Error = 1.971520
Iteration 4996: Error = 1.971515
Iteration 4997: Error = 1.971509
Iteration 4998: Error = 1.971504
Iteration 4999: Error = 1.971499
Iteration 5000: Error = 1.971493
Best Results:
Learning Rate: 0.1, Momentum: 0.2, Tolerance: 0.001
Total Iterations: 5000
Final Training Error: 1.971493

Error vs. Iterations Graph:
The error vs. iteration graph illustrates the convergence of the neural network during training. Initially, the squared error is high, but it monotonically decreases as the network learns from the training data. Rapidly fluctuations in the error indicate moments when the learning process adjusts



the weights, seeking for an optimal solution. At last, the error stabilizes, showing that the network has reached a nearer to optimal state.

Confusion Matrix:
The confusion matrix evaluates the performance of the trained neural network. This model accurately classifies almost all test samples, achieving nearly to perfect accuracy. Only one instance of the

"Versicolor" class is misclassified, while "Setosa" and "Virginica" are classified with almost 100% accuracy. Result indicates that the neural network distinguishes between the three classes of the Iris dataset.