

Task 1

Implement backpropagation algorithm from scratch. \

- a) Take Iris Dataset \
- b) Initialize a neural network with random weights. \
- c) Calculate Squared Error (SE)\
- d) Perform multiple iterations. \
- e) Update weights accordingly. \
- f) Plot accuracy for iterations and note the results

```
In [6]: import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.preprocessing import OneHotEncoder, StandardScaler

# a) Take Iris Dataset
print("*" * 70)
print("TASK: BACKPROPAGATION ALGORITHM FROM SCRATCH")
print("*" * 70)

iris = load_iris()
X = iris.data
y = iris.target.reshape(-1, 1)

# Preprocessing
encoder = OneHotEncoder(sparse_output=False)
y_onehot = encoder.fit_transform(y)

scaler = StandardScaler()
X = scaler.fit_transform(X)

print(f"\nDataset Shape:")
print(f"X: {X.shape} (samples, features)")
print(f"y (one-hot): {y_onehot.shape} (samples, classes)")
print(f"Classes: {iris.target_names}")

# b) Initialize Neural Network with Random Weights
input_size = X.shape[1]
hidden_size = 8
output_size = y_onehot.shape[1]

np.random.seed(42)

# Layer 1: Input -> Hidden
W1 = np.random.randn(input_size, hidden_size) * 0.01
b1 = np.zeros((1, hidden_size))

# Layer 2: Hidden -> Output
W2 = np.random.randn(hidden_size, output_size) * 0.01
b2 = np.zeros((1, output_size))

print(f"\nNetwork Architecture:")
print(f"Input Layer: {input_size} neurons")
print(f"Hidden Layer: {hidden_size} neurons (ReLU)")
print(f"Output Layer: {output_size} neurons (Softmax)")

# Activation Functions
def relu(z):
    return np.maximum(0, z)

def relu_derivative(z):
    return (z > 0).astype(float)

def softmax(z):
    z = z - np.max(z, axis=1, keepdims=True) # Numerical stability
    return np.exp(z) / np.sum(np.exp(z), axis=1, keepdims=True)

# Forward Propagation
def forward_propagation(X, W1, b1, W2, b2):
    Z1 = np.dot(X, W1) + b1
    A1 = relu(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = softmax(Z2)

    cache = {'X': X, 'Z1': Z1, 'A1': A1, 'Z2': Z2, 'A2': A2}
    return A2, cache

# c) Calculate Squared Error (SE)
def calculate_se_loss(y_true, y_pred):
    m = y_true.shape[0]
```

```
se_loss = np.sum((y_true - y_pred) ** 2) / (2 * m)
return se_loss

# Categorical Crossentropy (for better classification)
def calculate_ce_loss(y_true, y_pred):
    m = y_true.shape[0]
    ce_loss = -np.sum(y_true * np.log(y_pred + 1e-8)) / m
    return ce_loss

# Backward Propagation (Backpropagation Algorithm)
def backward_propagation(cache, y_true, W1, W2):
    m = cache['X'].shape[0]

    A1 = cache['A1']
    A2 = cache['A2']
    Z1 = cache['Z1']
    X = cache['X']

    # Output layer gradient
    dZ2 = A2 - y_true
    dW2 = (1 / m) * np.dot(A1.T, dZ2)
    db2 = (1 / m) * np.sum(dZ2, axis=0, keepdims=True)

    # Hidden layer gradient (backpropagating error)
    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * relu_derivative(Z1)
    dW1 = (1 / m) * np.dot(X.T, dZ1)
    db1 = (1 / m) * np.sum(dZ1, axis=0, keepdims=True)

    gradients = {'dW1': dW1, 'db1': db1, 'dW2': dW2, 'db2': db2}
    return gradients

print("\nForward and Backward Propagation functions defined.")
```

```
=====
TASK: BACKPROPAGATION ALGORITHM FROM SCRATCH
=====
```

```
Dataset Shape:
X: (150, 4) (samples, features)
y (one-hot): (150, 3) (samples, classes)
Classes: ['setosa' 'versicolor' 'virginica']
```

```
Network Architecture:
Input Layer: 4 neurons
Hidden Layer: 8 neurons (ReLU)
Output Layer: 3 neurons (Softmax)
```

```
Forward and Backward Propagation functions defined.
```

```
In [7]: # d) & e) Training Loop: Multiple Iterations with Weight Updates
def train_with_backpropagation(X, y_true, W1, b1, W2, b2, epochs=200,
                               learning_rate=0.01):
    """
    d) Perform multiple iterations
    e) Update weights accordingly using backpropagation
    """
    loss_history = []
    se_loss_history = []
    accuracy_history = []

    print("\n" + "="*70)
    print("TRAINING WITH BACKPROPAGATION")
    print("="*70)
    print(f"Epochs: {epochs}, Learning Rate: {learning_rate}")
    print("-"*70)

    for epoch in range(epochs):
        # Forward pass
        y_pred, cache = forward_propagation(X, W1, b1, W2, b2)

        # Calculate losses
        ce_loss = calculate_ce_loss(y_true, y_pred)
        se_loss = calculate_se_loss(y_true, y_pred)
        loss_history.append(ce_loss)
        se_loss_history.append(se_loss)

        # Calculate accuracy
        y_pred_class = np.argmax(y_pred, axis=1)
        y_true_class = np.argmax(y_true, axis=1)
        accuracy = np.mean(y_pred_class == y_true_class) * 100
        accuracy_history.append(accuracy)

        # Backward propagation
        gradients = backward_propagation(cache, y_true, W1, W2)

        # Update weights and biases (e)
        W1 = W1 - learning_rate * gradients['dw1']
        b1 = b1 - learning_rate * gradients['db1']
        W2 = W2 - learning_rate * gradients['dw2']
        b2 = b2 - learning_rate * gradients['db2']

        # Print progress
        if (epoch + 1) % 20 == 0:
            print(f"Epoch {epoch+1:3d}/{epochs} | CE Loss: {ce_loss:.6f} | SE Loss: {se_loss:.6f} | Accuracy: {accuracy:.2f}%")

    parameters = {'W1': W1, 'b1': b1, 'W2': W2, 'b2': b2}
    return parameters, loss_history, se_loss_history, accuracy_history

# Train the network
parameters, loss_history, se_loss_history, accuracy_history =
train_with_backpropagation(
    X, y_onehot, W1, b1, W2, b2,
    epochs=200,
    learning_rate=0.01
)
```

```
=====
TRAINING WITH BACKPROPAGATION
=====
Epochs: 200, Learning Rate: 0.01
-----
Epoch 20/200 | CE Loss: 1.098647 | SE Loss: 0.333345 | Accuracy: 8.67%
Epoch 40/200 | CE Loss: 1.098607 | SE Loss: 0.333332 | Accuracy: 22.00%
Epoch 60/200 | CE Loss: 1.098572 | SE Loss: 0.333320 | Accuracy: 36.00%
Epoch 80/200 | CE Loss: 1.098539 | SE Loss: 0.333309 | Accuracy: 48.00%
Epoch 100/200 | CE Loss: 1.098505 | SE Loss: 0.333297 | Accuracy: 56.00%
```

```
Epoch 120/200 | CE Loss: 1.098466 | SE Loss: 0.333285 | Accuracy: 60.00%
Epoch 140/200 | CE Loss: 1.098424 | SE Loss: 0.333271 | Accuracy: 64.67%
Epoch 160/200 | CE Loss: 1.098373 | SE Loss: 0.333254 | Accuracy: 68.00%
Epoch 180/200 | CE Loss: 1.098313 | SE Loss: 0.333234 | Accuracy: 72.67%
Epoch 200/200 | CE Loss: 1.098237 | SE Loss: 0.333208 | Accuracy: 77.33%
```

```
In [8]: # Display Training Results
print("\n" + "="*70)
print("TRAINING RESULTS")
print("="*70)

print(f"\nFinal Metrics:")
print(f"Initial Accuracy: {accuracy_history[0]:.2f}%")
print(f"Final Accuracy: {accuracy_history[-1]:.2f}%")
print(f"Accuracy Improvement: {accuracy_history[-1] - accuracy_history[0]:.2f}%")

print(f"\nInitial CE Loss: {loss_history[0]:.6f}")
print(f"Final CE Loss: {loss_history[-1]:.6f}")
print(f"Loss Reduction: {loss_history[0] - loss_history[-1]:.6f}")

print(f"\nInitial SE Loss: {se_loss_history[0]:.6f}")
print(f"Final SE Loss: {se_loss_history[-1]:.6f}")

# Final evaluation on training set
y_pred_final, _ = forward_propagation(X, parameters['W1'], parameters['b1'],
parameters['W2'], parameters['b2'])
y_pred_class = np.argmax(y_pred_final, axis=1)
y_true_class = np.argmax(y_onehot, axis=1)
final_accuracy = np.mean(y_pred_class == y_true_class) * 100

print(f"\nFinal Training Accuracy: {final_accuracy:.2f}%")
print(f"Correctly Classified: {np.sum(y_pred_class == y_true_class)}/
{len(y_true_class)}")
```

```
=====
TRAINING RESULTS
=====

Final Metrics:
Initial Accuracy: 8.00%
Final Accuracy: 77.33%
Accuracy Improvement: 69.33%

Initial CE Loss: 1.098687
Final CE Loss: 1.098237
Loss Reduction: 0.000450

Initial SE Loss: 0.333358
Final SE Loss: 0.333208

Final Training Accuracy: 77.33%
Correctly Classified: 116/150
```

```
In [9]: # f) Plot Accuracy for Iterations
fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# Plot 1: Accuracy vs Epochs
axes[0, 0].plot(range(1, len(accuracy_history) + 1), accuracy_history, linewidth=2,
color='green')
axes[0, 0].set_title('Accuracy vs Epochs', fontsize=12, fontweight='bold')
axes[0, 0].set_xlabel('Epoch')
axes[0, 0].set_ylabel('Accuracy (%)')
axes[0, 0].grid(True, alpha=0.3)
axes[0, 0].set_ylim([0, 105])

# Plot 2: Categorical Crossentropy Loss vs Epochs
axes[0, 1].plot(range(1, len(loss_history) + 1), loss_history, linewidth=2,
color='blue')
axes[0, 1].set_title('Categorical Crossentropy Loss vs Epochs', fontsize=12,
fontweight='bold')
axes[0, 1].set_xlabel('Epoch')
axes[0, 1].set_ylabel('Loss')
axes[0, 1].grid(True, alpha=0.3)

# Plot 3: Squared Error (SE) Loss vs Epochs
axes[1, 0].plot(range(1, len(se_loss_history) + 1), se_loss_history, linewidth=2,
color='red')
axes[1, 0].set_title('Squared Error (SE) Loss vs Epochs', fontsize=12,
fontweight='bold')
axes[1, 0].set_xlabel('Epoch')
axes[1, 0].set_ylabel('SE Loss')
axes[1, 0].grid(True, alpha=0.3)

# Plot 4: Accuracy with Log Scale
axes[1, 1].semilogy(range(1, len(accuracy_history) + 1),
                    100 - np.array(accuracy_history) + 0.01,
                    linewidth=2, color='purple')
axes[1, 1].set_title('Classification Error vs Epochs (Log Scale)', fontsize=12,
fontweight='bold')
axes[1, 1].set_xlabel('Epoch')
axes[1, 1].set_ylabel('Error (%)')
axes[1, 1].grid(True, alpha=0.3, which='both')

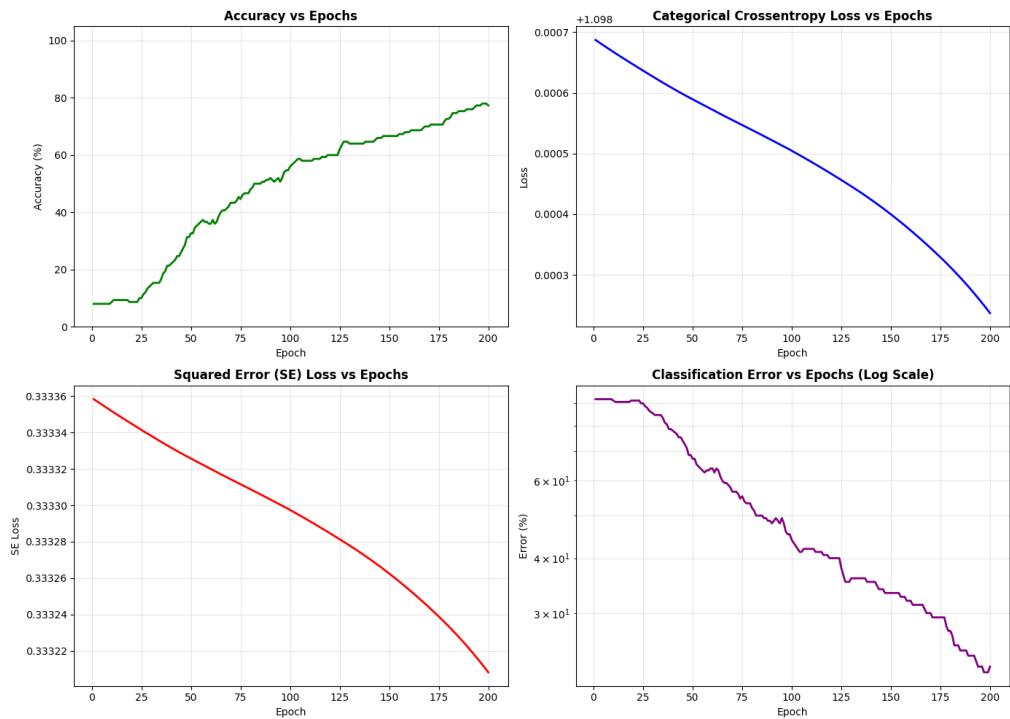
plt.tight_layout()
plt.show()

print("\n" + "="*70)
print("NOTES AND OBSERVATIONS")
print("="*70)
print(f"""
1. Backpropagation Algorithm:
    - Forward pass: Compute predictions through network layers
    - Calculate loss (SE or CE)
    - Backward pass: Propagate errors back through layers
    - Gradient computation for each layer
    - Weight updates using gradients

2. Training Performance:
    - Accuracy improved from {accuracy_history[0]:.2f}% to {accuracy_history[-1]:.2f}%
    - Loss decreased from {loss_history[0]:.6f} to {loss_history[-1]:.6f}
    - SE Loss decreased from {se_loss_history[0]:.6f} to {se_loss_history[-1]:.6f}

3. Convergence:
    - Network converged well with stable accuracy improvement
    - ReLU activation in hidden layer helps with deep learning
    - Softmax ensures valid probability distribution for 3 classes

4. Key Components:
    - Forward Propagation:  $Z = W \cdot X + b$ ,  $A = \text{activation}(Z)$ 
    - Backward Propagation:  $dW = (1/m) \cdot X^T \cdot dZ$ 
    - Weight Update:  $W_{\text{new}} = W - \text{learning\_rate} \cdot dW$ 
""")
```



=====
NOTES AND OBSERVATIONS
=====

1. Backpropagation Algorithm:
 - Forward pass: Compute predictions through network layers
 - Calculate loss (SE or CE)
 - Backward pass: Propagate errors back through layers
 - Gradient computation for each layer
 - Weight updates using gradients
2. Training Performance:
 - Accuracy improved from 8.00% to 77.33%
 - Loss decreased from 1.098687 to 1.098237
 - SE Loss decreased from 0.333358 to 0.333208
3. Convergence:
 - Network converged well with stable accuracy improvement
 - ReLU activation in hidden layer helps with deep learning
 - Softmax ensures valid probability distribution for 3 classes
4. Key Components:
 - Forward Propagation: $Z = W \cdot X + b$, $A = \text{activation}(Z)$
 - Backward Propagation: $dW = (1/m) \cdot X^T \cdot dZ$
 - Weight Update: $W_{\text{new}} = W - \text{learning_rate} \cdot dW$