Neural Network Library Documentation

Overview

This documentation provides an overview of a neural network library, covering the main components required to build, train, and evaluate a neural network. The library includes classes for data loading, network layers, activation functions, dropout, loss functions, and optimizers.

Network

Class: `Network`

The `Network` class represents the neural network, managing layers, forward and backward passes, and the overall training and evaluation process.

- **Constructor: `__init__(self, loss, optimizer)`**
  - **Parameters**:
    - `loss` (class): Loss function class (e.g., `MSELoss`, `MAELoss`, `BCELoss`).
    - `optimizer` (object): Optimizer instance (e.g., `SGD`, `Adadelta`, `Adam`).
- **Method: `train_mode(self)`**
  - **Description**: Sets the network to training mode.
- **Method: `evaluation_mode(self)`**
  - **Description**: Sets the network to evaluation mode.
- **Method: `add_layer(self, layer, layer_inputs, layer_outputs, distribution="normal")`**
  - **Description**: Adds a layer to the network.
  - **Parameters**:
    - `layer` (class): Layer class (e.g., `Linear`).
    - `layer_inputs` (int): Number of input neurons to the layer.
    - `layer_outputs` (int): Number of output neurons from the layer.
    - `distribution` (str): Distribution for weight initialization. Default is `"normal"`.
- **Method: `add_dropout(self, layer_inputs, fraction=0.5)`**
  - **Description**: Adds a dropout layer to the network.
  - **Parameters**:
    - `layer_inputs` (int): Number of input neurons to the layer.
    - `fraction` (float): Fraction of the input units to drop. Default is 0.5.
- **Method: `add_activation(self, activation)`**
  - **Description**: Adds an activation function to the network.
  - **Parameters**:
    - `activation` (object): Activation function instance (e.g., `Sigmoid`, `Tanh`, `Relu`, `LeakyRelu`).
- **Method: `evaluate(self, x)`**
  - **Description**: Performs a forward pass through the network for evaluation.
  - **Parameters**:
    - `x` (numpy array): Input data.
  - **Returns**: Output of the network.
- **Method: `train(self, x, y)`**
  - **Description**: Performs a forward and backward pass through the network for training.
  - **Parameters**:
    - `x` (numpy array): Input data.
    - `y` (numpy array): Correct labels.
  - **Returns**: Mean loss of the batch.

Example Usage

```
import numpy as np
from Network import Network
from Linear import Linear
from ActivationFunctions import Relu, Sigmoid
from Dropout import Dropout
from LossFunctions import MSELoss
from Optimizers import Adam

# Example input and labels
```

```
x = np.random.rand(10, 5)
y = np.random.rand(10, 1)

# Initialize optimizer
optimizer = Adam(eta=0.001)

# Initialize network
network = Network(loss=MSELoss, optimizer=optimizer)

# Add layers
network.add_layer(Linear, layer_inputs=5, layer_outputs=10, distribution="normal_xavier")
network.add_activation(Relu())
network.add_dropout(layer_inputs=10, fraction=0.5)
network.add_layer(Linear, layer_inputs=10, layer_outputs=1, distribution="normal_xavier")
network.add_activation(Sigmoid())

# Set to training mode
network.train_mode()

# Train the network
loss = network.train(x, y)
print("Training Loss:", loss)

# Set to evaluation mode
network.evaluation_mode()

# Evaluate the network
output = network.evaluate(x)
print("Evaluation Output:", output)
```

Data Loader

Class: `DataLoader`

The `DataLoader` class handles loading and iterating over datasets for training and evaluation.

- **Constructor: `__init__(self, inputs, labels, batch_size=1, shuffle=False, drop_last=False)`**
  - **Parameters**:
    - `inputs` (numpy array): Input data.
    - `labels` (numpy array): Corresponding labels.
    - `batch_size` (int): Size of each data batch. Default is 1.
    - `shuffle` (bool): Whether to shuffle the data before each epoch. Default is `False`.
    - `drop_last` (bool): Whether to drop the last batch if it's smaller than the batch size. Default is `False`.
- **Method: `__iter__(self)`**
  - **Description**: Creates an iterator over the data.
  - **Returns**: Batches of inputs and labels.

Example Usage

```
import numpy as np
from DataLoader import DataLoader

# Example input and labels
inputs = np.random.rand(100, 20)
labels = np.random.rand(100, 1)

# Initialize data loader
data_loader = DataLoader(inputs, labels, batch_size=10, shuffle=True)

# Iterate over batches
for batch_inputs, batch_labels in data_loader:
```

```
print("Batch inputs:", batch_inputs)
print("Batch labels:", batch_labels)
```

Layers

Class: `Linear`

The `Linear` class defines a fully connected layer in the neural network.

- **Constructor: `__init__(self, input_neurons, output_neurons, optimizer, distribution)`**
  - **Parameters**:
    - `input_neurons` (int): Number of input neurons.
    - `output_neurons` (int): Number of output neurons.
    - `optimizer` (object): Optimizer instance (e.g., `SGD`, `Adadelta`, `Adam`).
    - `distribution` (str): Distribution for weight initialization (e.g., `"normal_xavier"`, `"uniform_xavier"`, `"normal"`).
- **Method: `forward(self, x)`**
  - **Description**: Performs the forward pass.
  - **Parameters**:
    - `x` (numpy array): Input data.
  - **Returns**: Output of the layer.
- **Method: `backward(self, gradient, layer_input)`**
  - **Description**: Performs the backward pass.
  - **Parameters**:
    - `gradient` (numpy array): Gradient from the next layer.
    - `layer_input` (numpy array): Input data to the layer during forward pass.
  - **Returns**: Gradient for the previous layer.

Example Usage

```
import numpy as np
from Linear import Linear
from Optimizers import Adam

# Example input
x = np.random.rand(10, 5)

# Initialize optimizer
optimizer = Adam(eta=0.001)

# Initialize linear layer
linear_layer = Linear(input_neurons=5, output_neurons=3, optimizer=optimizer, distribution="normal_xavier")

# Forward pass
output = linear_layer.forward(x)
print("Layer output:", output)

# Backward pass
gradient = np.random.rand(10, 3)
backprop_gradient = linear_layer.backward(gradient, x)
print("Backpropagation gradient:", backprop_gradient)
```

Activation Functions

Class: `Sigmoid`

The `Sigmoid` class implements the sigmoid activation function.

- **Method: `forward(self, layer_input)`**
  - **Description**: Applies the sigmoid function.
  - **Parameters**:
    - `layer_input` (numpy array): Input data.

  - **Returns**: Activated output.
- **Method: `backward(self, gradient, layer_input)`**
  - **Description**: Computes the gradient of the sigmoid function.
  - **Parameters**:
    - `gradient` (numpy array): Gradient from the next layer.
    - `layer_input` (numpy array): Input data to the layer during forward pass.
  - **Returns**: Gradient for the previous layer.

Class: `Tanh`

The `Tanh` class implements the tanh activation function.

- **Method: `forward(self, layer_input)`**
  - **Description**: Applies the tanh function.
  - **Parameters**:
    - `layer_input` (numpy array): Input data.
  - **Returns**: Activated output.
- **Method: `backward(self, gradient, layer_input)`**
  - **Description**: Computes the gradient of the tanh function.
  - **Parameters**:
    - `gradient` (numpy array): Gradient from the next layer.
    - `layer_input` (numpy array): Input data to the layer during forward pass.
  - **Returns**: Gradient for the previous layer.

Class: `Relu`

The `Relu` class implements the ReLU activation function.

- **Method: `forward(self, layer_input)`**
  - **Description**: Applies the ReLU function.
  - **Parameters**:
    - `layer_input` (numpy array): Input data.
  - **Returns**: Activated output.
- **Method: `backward(self, gradient, layer_input)`**
  - **Description**: Computes the gradient of the ReLU function.
  - **Parameters**:
    - `gradient` (numpy array): Gradient from the next layer.
    - `layer_input` (numpy array): Input data to the layer during forward pass.
  - **Returns**: Gradient for the previous layer

.

Class: `LeakyRelu`

The `LeakyRelu` class implements the leaky ReLU activation function.

- **Constructor: `__init__(self, a=0.01)`**
  - **Parameters**:
    - `a` (float): Slope of the activation function for negative inputs. Default is 0.01.
- **Method: `forward(self, layer_input)`**
  - **Description**: Applies the leaky ReLU function.
  - **Parameters**:
    - `layer_input` (numpy array): Input data.
  - **Returns**: Activated output.
- **Method: `backward(self, gradient, layer_input)`**
  - **Description**: Computes the gradient of the leaky ReLU function.
  - **Parameters**:
    - `gradient` (numpy array): Gradient from the next layer.
    - `layer_input` (numpy array): Input data to the layer during forward pass.
  - **Returns**: Gradient for the previous layer.

Example Usage

```
import numpy as np
from ActivationFunctions import Sigmoid, Tanh, Relu, LeakyRelu

# Example input
x = np.random.rand(10, 5)

# Sigmoid activation
sigmoid = Sigmoid()
sigmoid_output = sigmoid.forward(x)
print("Sigmoid output:", sigmoid_output)

# Tanh activation
tanh = Tanh()
tanh_output = tanh.forward(x)
print("Tanh output:", tanh_output)

# ReLU activation
relu = Relu()
relu_output = relu.forward(x)
print("ReLU output:", relu_output)

# Leaky ReLU activation
leaky_relu = LeakyRelu(a=0.1)
leaky_relu_output = leaky_relu.forward(x)
print("Leaky ReLU output:", leaky_relu_output)
```

Dropout

Class: `Dropout`

The `Dropout` class implements dropout regularization.

- **Constructor: `__init__(self, input_neurons, fraction, training_mode)`**
  - **Parameters**:
    - `input_neurons` (int): Number of input neurons.
    - `fraction` (float): Fraction of the input units to drop.
    - `training_mode` (list): List containing a single boolean indicating training mode.
- **Method: `forward(self, layer_input)`**
  - **Description**: Applies dropout during training.
  - **Parameters**:
    - `layer_input` (numpy array): Input data.
  - **Returns**: Output after applying dropout.
- **Method: `backward(self, gradient, *_)`**
  - **Description**: Computes the gradient for dropout.
  - **Parameters**:
    - `gradient` (numpy array): Gradient from the next layer.
  - **Returns**: Gradient for the previous layer.

Example Usage

```
import numpy as np
from Dropout import Dropout

# Example input
x = np.random.rand(10, 5)

# Initialize dropout
training_mode = [True]
dropout = Dropout(input_neurons=5, fraction=0.5, training=training_mode)
```

```python
# Forward pass (training mode)
dropout_output = dropout.forward(x)
print("Dropout output (training):", dropout_output)

# Set to evaluation mode
dropout.training[0] = False
dropout_output_eval = dropout.forward(x)
print("Dropout output (evaluation):", dropout_output_eval)
```

Loss Functions

Class: `MSELoss`

The `MSELoss` class implements the Mean Squared Error loss function.

- **Method:** `loss(self, y_pred, y_true)`
  - **Description**: Computes the MSE loss.
  - **Parameters**:
    - `y_pred` (numpy array): Predicted values.
    - `y_true` (numpy array): True values.
  - **Returns**: MSE loss.
- **Method:** `loss_gradient(self, y_pred, y_true)`
  - **Description**: Computes the gradient of the MSE loss.
  - **Parameters**:
    - `y_pred` (numpy array): Predicted values.
    - `y_true` (numpy array): True values.
  - **Returns**: Gradient of the MSE loss.

Class: `MAELoss`

The `MAELoss` class implements the Mean Absolute Error loss function.

- **Method:** `loss(self, y_pred, y_true)`
  - **Description**: Computes the MAE loss.
  - **Parameters**:
    - `y_pred` (numpy array): Predicted values.
    - `y_true` (numpy array): True values.
  - **Returns**: MAE loss.
- **Method:** `loss_gradient(self, y_pred, y_true)`
  - **Description**: Computes the gradient of the MAE loss.
  - **Parameters**:
    - `y_pred` (numpy array): Predicted values.
    - `y_true` (numpy array): True values.
  - **Returns**: Gradient of the MAE loss.

Class: `BCELoss`

The `BCELoss` class implements the Binary Cross-Entropy loss function.

- **Method:** `loss(self, y_pred, y_true)`
  - **Description**: Computes the BCE loss.
  - **Parameters**:
    - `y_pred` (numpy array): Predicted values.
    - `y_true` (numpy array): True values.
  - **Returns**: BCE loss.
- **Method:** `loss_gradient(self, y_pred, y_true)`
  - **Description**: Computes the gradient of the BCE loss.
  - **Parameters**:
    - `y_pred` (numpy array): Predicted values.
    - `y_true` (numpy array): True values.
  - **Returns**: Gradient of the BCE loss.

Example Usage

```
import numpy as np
from LossFunctions import MSELoss, MAELoss, BCELoss

# Example predictions and true values
y_pred = np.random.rand(10, 1)
y_true = np.random.rand(10, 1)

# MSE Loss
mse_loss = MSELoss()
loss_value = mse_loss.loss(y_pred, y_true)
loss_gradient = mse_loss.loss_gradient(y_pred, y_true)
print("MSE Loss value:", loss_value)
print("MSE Loss gradient:", loss_gradient)

# MAE Loss
mae_loss = MAELoss()
loss_value = mae_loss.loss(y_pred, y_true)
loss_gradient = mae_loss.loss_gradient(y_pred, y_true)
print("MAE Loss value:", loss_value)
print("MAE Loss gradient:", loss_gradient)

# BCE Loss
bce_loss = BCELoss()
loss_value = bce_loss.loss(y_pred, y_true)
loss_gradient = bce_loss.loss_gradient(y_pred, y_true)
print("BCE Loss value:", loss_value)
print("BCE Loss gradient:", loss_gradient)
```

Optimizers

Class: `SGD`

The `SGD` class implements the Stochastic Gradient Descent optimizer.

- **Constructor: `__init__(self, learning_rate=0.001, momentum=0, nesterov=False)`**
  - **Parameters**:
    - `learning_rate` (float): Learning rate. Default is 0.001.
    - `momentum` (float): Momentum factor. Default is 0.
    - `nesterov` (bool): Whether to use Nesterov momentum. Default is `False`.
- **Method: `optimize(self, weights, biases, gradient_weights, gradient_biases)`**
  - **Description**: Updates weights and biases based on gradients.
  - **Parameters**:
    - `weights` (numpy array): Current weights.
    - `biases` (numpy array): Current biases.
    - `gradient_weights` (numpy array): Gradients of weights.
    - `gradient_biases` (numpy array): Gradients of biases.
  - **Returns**: Updated weights and biases.

Class: `Adadelta`

The `Adadelta` class implements the Adadelta optimizer.

- **Constructor: `__init__(self, gamma=0.9, eps=1e-8)`**
  - **Parameters**:
    - `gamma` (float): Decay rate. Default is 0.9.
    - `eps` (float): Small constant to prevent division by zero. Default is 1e-8.
- **Method: `optimize(self, weights, biases, gradient_weights, gradient_biases)`**
  - **Description**: Updates weights and biases based on gradients.
  - **Parameters**:
    - `weights` (numpy array): Current weights.

        * `biases` (numpy array): Current biases.
        * `gradient_weights` (numpy array): Gradients of weights.
        * `gradient_biases` (numpy array): Gradients of biases.
    - **Returns**: Updated weights and biases.

Class: `Adam`

The `Adam` class implements the Adam optimizer.

- **Constructor: `__init__(self, eta=0.001, beta1=0.9, beta2=0.999, eps=1e-8)`**
    - **Parameters**:
        * `eta` (float): Learning rate. Default is 0.001.
        * `beta1` (float): Exponential decay rate for the first moment estimates. Default is 0.9.
        * `beta2` (float): Exponential decay rate for the second moment estimates. Default is 0.999.
        * `eps` (float): Small

constant to prevent division by zero. Default is 1e-8.

- **Method: `optimize(self, weights, biases, gradient_weights, gradient_biases)`**
    - **Description**: Updates weights and biases based on gradients.
    - **Parameters**:
        * `weights` (numpy array): Current weights.
        * `biases` (numpy array): Current biases.
        * `gradient_weights` (numpy array): Gradients of weights.
        * `gradient_biases` (numpy array): Gradients of biases.
    - **Returns**: Updated weights and biases.

Example Usage

```
import numpy as np
from Optimizers import SGD, Adadelta, Adam

# Example weights and gradients
weights = np.random.rand(3, 2)
biases = np.random.rand(2)
gradient_weights = np.random.rand(3, 2)
gradient_biases = np.random.rand(2)

# SGD Optimizer
sgd = SGD(learning_rate=0.01, momentum=0.9)
updated_weights, updated_biases = sgd.optimize(weights, biases, gradient_weights, gradient_biases)
print("SGD updated weights:", updated_weights)
print("SGD updated biases:", updated_biases)

# Adadelta Optimizer
adadelta = Adadelta(gamma=0.9)
updated_weights, updated_biases = adadelta.optimize(weights, biases, gradient_weights, gradient_biases)
print("Adadelta updated weights:", updated_weights)
print("Adadelta updated biases:", updated_biases)

# Adam Optimizer
adam = Adam(eta=0.001)
updated_weights, updated_biases = adam.optimize(weights, biases, gradient_weights, gradient_biases)
print("Adam updated weights:", updated_weights)
print("Adam updated biases:", updated_biases)
```