DnnLib User Manual

A high-performance C++ neural network library with Python bindings, built on the xtensor ecosystem for efficient numerical computing.

Table of Contents

- Installation
- Quick Start
- Core Components
  - Dense Layers
  - Activation Functions
  - Loss Functions
  - Optimizers
- Training a Neural Network
- Complete Examples
- API Reference

Installation

Prerequisites

- Python 3.8+
- NumPy
- CMake 3.14+
- C++17 compatible compiler

Import the Library

```
import sys
import DnnLib
import numpy as np
```

Quick Start

Here's a simple example to get you started:

```
import DnnLib
import numpy as np

# Create sample data (XOR problem)
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=np.float64)
y = np.array([[0], [1], [1], [0]], dtype=np.float64)

# Create a neural network: 2 -> 4 -> 1
layer1 = DnnLib.DenseLayer(2, 4, DnnLib.ActivationType.RELU)
layer2 = DnnLib.DenseLayer(4, 1, DnnLib.ActivationType.SIGMOID)
```

```python
# Create optimizer
optimizer = DnnLib.Adam(learning_rate=0.01)

# Training loop
for epoch in range(100):
    # Forward pass
    h1 = layer1.forward(X)
    output = layer2.forward(h1)

    # Compute loss
    loss = DnnLib.mse(output, y)

    # Backward pass
    loss_grad = DnnLib.mse_gradient(output, y)
    grad2 = layer2.backward(loss_grad)
    grad1 = layer1.backward(grad2)

    # Update parameters
    optimizer.update(layer2)
    optimizer.update(layer1)

    if epoch % 20 == 0:
        print(f"Epoch {epoch}, Loss: {loss:.6f}")
```

Core Components

Dense Layers

Dense (fully connected) layers are the core building blocks of neural networks.

Creating Dense Layers

```python
# Basic dense layer: input_dim=3, output_dim=5, default ReLU activation
layer = DnnLib.DenseLayer(3, 5)

# With specific activation function
layer = DnnLib.DenseLayer(3, 5, DnnLib.ActivationType.SIGMOID)
```

Available Activation Types

- `DnnLib.ActivationType.RELU` - Rectified Linear Unit (default)
- `DnnLib.ActivationType.SIGMOID` - Sigmoid function
- `DnnLib.ActivationType.TANH` - Hyperbolic tangent
- `DnnLib.ActivationType.SOFTMAX` - Softmax (for multi-class output)

Layer Operations

```python
# Forward pass (supports both single samples and batches)
single_input = np.array([1.0, 2.0, 3.0], dtype=np.float64)
batch_input = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]], dtype=np.float64)
```

```
output = layer.forward(single_input)  # Shape: (5,)
batch_output = layer.forward(batch_input)  # Shape: (2, 5)

# Forward pass without activation (linear only)
linear_output = layer.forward_linear(single_input)

# Backward pass (for gradient computation)
gradient_input = np.ones(5, dtype=np.float64)  # Gradient from next layer
input_gradient = layer.backward(gradient_input)

# Access parameters
weights = layer.weights  # Shape: (output_dim, input_dim)
bias = layer.bias        # Shape: (output_dim,)

# Access gradients (after backward pass)
weight_grads = layer.weight_gradients
bias_grads = layer.bias_gradients

# Modify activation function
layer.activation_type = DnnLib.ActivationType.TANH
```

Activation Functions

DnnLib provides standalone activation functions for custom use:

```
x = np.array([-2, -1, 0, 1, 2], dtype=np.float64)

# Individual activation functions
sigmoid_out = DnnLib.sigmoid(x)
tanh_out = DnnLib.tanh(x)
relu_out = DnnLib.relu(x)
softmax_out = DnnLib.softmax(x)  # For probability distributions

# Derivative functions (for custom backpropagation)
sigmoid_deriv = DnnLib.sigmoid_derivative(x)
tanh_deriv = DnnLib.tanh_derivative(x)
relu_deriv = DnnLib.relu_derivative(x)

# Generic activation function
output = DnnLib.apply_activation(x, DnnLib.ActivationType.RELU)
derivative = DnnLib.apply_activation_derivative(x, DnnLib.ActivationType.RELU)
```

Loss Functions

DnnLib supports multiple loss functions for different types of problems:

Regression Loss Functions

```
y_pred = np.array([0.8, 0.9, 0.7], dtype=np.float64)
y_true = np.array([1.0, 1.0, 0.5], dtype=np.float64)
```

```
# Mean Squared Error
mse_loss = DnnLib.mse(y_pred, y_true)
mse_grad = DnnLib.mse_gradient(y_pred, y_true)

# Mean Absolute Error
mae_loss = DnnLib.mae(y_pred, y_true)
mae_grad = DnnLib.mae_gradient(y_pred, y_true)

# Huber Loss (robust to outliers)
huber_loss = DnnLib.huber(y_pred, y_true, delta=1.0)
huber_grad = DnnLib.huber_gradient(y_pred, y_true, delta=1.0)
```

Classification Loss Functions

```
# For multi-class classification (with softmax output)
y_pred = np.array([0.1, 0.7, 0.2], dtype=np.float64)  # Probabilities
y_true = np.array([0.0, 1.0, 0.0], dtype=np.float64)  # One-hot encoded

cross_entropy_loss = DnnLib.cross_entropy(y_pred, y_true)
cross_entropy_grad = DnnLib.cross_entropy_gradient(y_pred, y_true)

# For binary classification (with sigmoid output)
y_pred_binary = np.array([0.3, 0.7, 0.2], dtype=np.float64)
y_true_binary = np.array([0.0, 1.0, 0.0], dtype=np.float64)

binary_ce_loss = DnnLib.binary_cross_entropy(y_pred_binary, y_true_binary)
binary_ce_grad = DnnLib.binary_cross_entropy_gradient(y_pred_binary, y_true_binary)
```

Generic Loss Interface

```
# Using generic loss functions
loss_types = [
    DnnLib.LossType.MSE,
    DnnLib.LossType.MAE,
    DnnLib.LossType.HUBER,
    DnnLib.LossType.CROSS_ENTROPY,
    DnnLib.LossType.BINARY_CROSS_ENTROPY
]

for loss_type in loss_types:
    loss = DnnLib.compute_loss(y_pred, y_true, loss_type)
    gradient = DnnLib.compute_loss_gradient(y_pred, y_true, loss_type)
    print(f"{loss_type.name}: Loss = {loss:.4f}")
```

Optimizers

DnnLib provides several state-of-the-art optimization algorithms:

Stochastic Gradient Descent (SGD)

```
# Basic SGD
optimizer = DnnLib.SGD(learning_rate=0.01)
```

```python
# SGD with momentum
optimizer = DnnLib.SGD(learning_rate=0.01, momentum=0.9)

# Properties
print(f"Learning rate: {optimizer.learning_rate}")
print(f"Momentum: {optimizer.momentum}")

# Modify parameters
optimizer.learning_rate = 0.001
optimizer.momentum = 0.95
```

Adam Optimizer

```python
# Adam with default parameters
optimizer = DnnLib.Adam()

# Adam with custom parameters
optimizer = DnnLib.Adam(
    learning_rate=0.001,
    beta1=0.9,         # Momentum decay rate
    beta2=0.999,       # RMSprop decay rate
    epsilon=1e-8       # Numerical stability
)

# Access and modify parameters
optimizer.learning_rate = 0.01
optimizer.beta1 = 0.95
optimizer.beta2 = 0.9999
optimizer.epsilon = 1e-7
```

RMSprop Optimizer

```python
# RMSprop with default parameters
optimizer = DnnLib.RMSprop()

# RMSprop with custom parameters
optimizer = DnnLib.RMSprop(
    learning_rate=0.001,
    decay_rate=0.9,    # Moving average decay rate
    epsilon=1e-8       # Numerical stability
)

# Modify parameters
optimizer.decay_rate = 0.95
optimizer.epsilon = 1e-6
```

Optimizer Operations

```python
# All optimizers support these operations:

# Update layer parameters (call after backward pass)
optimizer.update(layer)
```

```
# Reset optimizer state (clear momentum/adaptive terms)
optimizer.reset()

# Access learning rate (common to all optimizers)
current_lr = optimizer.learning_rate
optimizer.learning_rate = 0.001
```

Optimizer Factory

```
# Create optimizers using factory function
sgd_opt = DnnLib.create_optimizer(
    DnnLib.OptimizerType.SGD_TYPE,
    learning_rate=0.01,
    param1=0.9  # momentum
)

adam_opt = DnnLib.create_optimizer(
    DnnLib.OptimizerType.ADAM_TYPE,
    learning_rate=0.001,
    param1=0.9,    # beta1
    param2=0.999,  # beta2
    param3=1e-8    # epsilon
)
```

Training a Neural Network

Basic Training Loop Structure

```
def train_network(layers, optimizer, X, y, epochs=100,
loss_type=DnnLib.LossType.MSE):
    """
    Generic training function for a neural network

    Args:
        layers: List of DenseLayer objects
        optimizer: Optimizer instance
        X: Input data (samples x features)
        y: Target data (samples x outputs)
        epochs: Number of training epochs
        loss_type: Type of loss function to use
    """
    for epoch in range(epochs):
        # Forward pass
        activations = [X]
        for layer in layers:
            activations.append(layer.forward(activations[-1]))

        output = activations[-1]

        # Compute loss
        loss = DnnLib.compute_loss(output, y, loss_type)
```

```
        # Backward pass
        grad = DnnLib.compute_loss_gradient(output, y, loss_type)

        for i in reversed(range(len(layers))):
            grad = layers[i].backward(grad)
            optimizer.update(layers[i])

        if epoch % (epochs // 10) == 0:
            print(f"Epoch {epoch}, Loss: {loss:.6f}")

    return loss
```

Mini-batch Training

```
def train_minibatch(layers, optimizer, X, y, batch_size=32, epochs=100):
    """
    Training with mini-batches for better performance
    """
    n_samples = X.shape[0]

    for epoch in range(epochs):
        # Shuffle data
        indices = np.random.permutation(n_samples)
        X_shuffled = X[indices]
        y_shuffled = y[indices]

        epoch_loss = 0.0
        n_batches = 0

        # Process mini-batches
        for i in range(0, n_samples, batch_size):
            X_batch = X_shuffled[i:i+batch_size]
            y_batch = y_shuffled[i:i+batch_size]

            # Forward pass
            activations = [X_batch]
            for layer in layers:
                activations.append(layer.forward(activations[-1]))

            # Loss and backward pass
            loss = DnnLib.mse(activations[-1], y_batch)
            grad = DnnLib.mse_gradient(activations[-1], y_batch)

            for j in reversed(range(len(layers))):
                grad = layers[j].backward(grad)
                optimizer.update(layers[j])

            epoch_loss += loss
            n_batches += 1

        avg_loss = epoch_loss / n_batches
        if epoch % 10 == 0:
```

```
            print(f"Epoch {epoch}, Avg Loss: {avg_loss:.6f}")
```

Complete Examples

1. Binary Classification

```
import DnnLib
import numpy as np

# Generate binary classification data
np.random.seed(42)
X = np.random.randn(1000, 2).astype(np.float64)
y = (X[:, 0] + X[:, 1] > 0).astype(np.float64).reshape(-1, 1)

print(f"Dataset: {X.shape[0]} samples, {X.shape[1]} features")
print(f"Positive class: {np.sum(y)}/{len(y)} samples")

# Create network: 2 -> 8 -> 4 -> 1
layers = [
    DnnLib.DenseLayer(2, 8, DnnLib.ActivationType.RELU),
    DnnLib.DenseLayer(8, 4, DnnLib.ActivationType.RELU),
    DnnLib.DenseLayer(4, 1, DnnLib.ActivationType.SIGMOID)
]

# Use Adam optimizer
optimizer = DnnLib.Adam(learning_rate=0.01)

# Training loop
print("Training binary classifier...")
for epoch in range(200):
    # Forward pass
    h1 = layers[0].forward(X)
    h2 = layers[1].forward(h1)
    output = layers[2].forward(h2)

    # Binary cross-entropy loss
    loss = DnnLib.binary_cross_entropy(output, y)

    # Backward pass
    grad = DnnLib.binary_cross_entropy_gradient(output, y)
    grad = layers[2].backward(grad)
    grad = layers[1].backward(grad)
    grad = layers[0].backward(grad)

    # Update all layers
    optimizer.update(layers[2])
    optimizer.update(layers[1])
    optimizer.update(layers[0])

    if epoch % 40 == 0:
        # Calculate accuracy
        predictions = (output > 0.5).astype(np.float64)
```

```
        accuracy = np.mean(predictions == y)
        print(f"Epoch {epoch}, Loss: {loss:.4f}, Accuracy: {accuracy:.4f}")

print("Training completed!")
```

2. Multi-class Classification

```
import DnnLib
import numpy as np

# Generate 3-class classification data
np.random.seed(42)
n_samples = 1500
n_features = 4
n_classes = 3

X = np.random.randn(n_samples, n_features).astype(np.float64)

# Create 3 classes with different patterns
y_labels = np.zeros(n_samples, dtype=int)
y_labels[:500] = 0  # Class 0
y_labels[500:1000] = 1  # Class 1
y_labels[1000:] = 2  # Class 2

# Convert to one-hot encoding
y = np.zeros((n_samples, n_classes), dtype=np.float64)
y[np.arange(n_samples), y_labels] = 1.0

print(f"Multi-class dataset: {X.shape}, {n_classes} classes")

# Create network: 4 -> 16 -> 8 -> 3
layers = [
    DnnLib.DenseLayer(4, 16, DnnLib.ActivationType.RELU),
    DnnLib.DenseLayer(16, 8, DnnLib.ActivationType.RELU),
    DnnLib.DenseLayer(8, 3, DnnLib.ActivationType.SOFTMAX)
]

# Use RMSprop optimizer
optimizer = DnnLib.RMSprop(learning_rate=0.01, decay_rate=0.9)

print("Training multi-class classifier...")
for epoch in range(150):
    # Forward pass
    h1 = layers[0].forward(X)
    h2 = layers[1].forward(h1)
    output = layers[2].forward(h2)

    # Cross-entropy loss
    loss = DnnLib.cross_entropy(output, y)

    # Backward pass
    grad = DnnLib.cross_entropy_gradient(output, y)
    grad = layers[2].backward(grad)
```

```
    grad = layers[1].backward(grad)
    grad = layers[0].backward(grad)

    # Update parameters
    for layer in layers:
        optimizer.update(layer)

    if epoch % 30 == 0:
        # Calculate accuracy
        predicted_classes = np.argmax(output, axis=1)
        accuracy = np.mean(predicted_classes == y_labels)
        print(f"Epoch {epoch}, Loss: {loss:.4f}, Accuracy: {accuracy:.4f}")

print("Multi-class training completed!")
```

3. Regression Problem

```
import DnnLib
import numpy as np

# Generate regression data: y = x₁² + x₂² + noise
np.random.seed(42)
n_samples = 2000
X = np.random.uniform(-2, 2, size=(n_samples, 2)).astype(np.float64)
y = (X[:, 0]**2 + X[:, 1]**2 + 0.1 * np.random.randn(n_samples)).reshape(-1, 1)

print(f"Regression dataset: {X.shape} -> {y.shape}")
print(f"Target range: [{np.min(y):.2f}, {np.max(y):.2f}]")

# Create deeper network for regression: 2 -> 32 -> 16 -> 8 -> 1
layers = [
    DnnLib.DenseLayer(2, 32, DnnLib.ActivationType.RELU),
    DnnLib.DenseLayer(32, 16, DnnLib.ActivationType.RELU),
    DnnLib.DenseLayer(16, 8, DnnLib.ActivationType.RELU),
    DnnLib.DenseLayer(8, 1, DnnLib.ActivationType.RELU)  # No activation for
regression output
]

# Try different optimizers
optimizers = [
    ("SGD", DnnLib.SGD(0.001)),
    ("SGD+Momentum", DnnLib.SGD(0.001, 0.9)),
    ("Adam", DnnLib.Adam(0.001)),
    ("RMSprop", DnnLib.RMSprop(0.001))
]

for opt_name, optimizer in optimizers:
    print(f"\n--- Training with {opt_name} ---")

    # Reset network weights (create new layers)
    layers = [
        DnnLib.DenseLayer(2, 32, DnnLib.ActivationType.RELU),
        DnnLib.DenseLayer(32, 16, DnnLib.ActivationType.RELU),
```

```
        DnnLib.DenseLayer(16, 8, DnnLib.ActivationType.RELU),
        DnnLib.DenseLayer(8, 1, DnnLib.ActivationType.RELU)
    ]
    optimizer.reset()

    # Training
    for epoch in range(100):
        # Forward pass
        activation = X
        for layer in layers:
            activation = layer.forward(activation)
        output = activation

        # MSE loss for regression
        loss = DnnLib.mse(output, y)

        # Backward pass
        grad = DnnLib.mse_gradient(output, y)
        for layer in reversed(layers):
            grad = layer.backward(grad)
            optimizer.update(layer)

        if epoch % 25 == 0:
            # Calculate R² score
            ss_res = np.sum((y - output) ** 2)
            ss_tot = np.sum((y - np.mean(y)) ** 2)
            r2_score = 1 - (ss_res / ss_tot)
            print(f"  Epoch {epoch}, Loss: {loss:.6f}, R²: {r2_score:.4f}")

    print(f"  Final {opt_name} Loss: {loss:.6f}")
```

API Reference

DenseLayer Class

```
class DenseLayer:
    def __init__(self, input_dim: int, output_dim: int,
                 activation_type: ActivationType = ActivationType.RELU)

    def forward(self, x: np.ndarray) -> np.ndarray
    def forward_linear(self, x: np.ndarray) -> np.ndarray
    def backward(self, gradient_output: np.ndarray) -> np.ndarray

    # Properties
    weights: np.ndarray          # Shape: (output_dim, input_dim)
    bias: np.ndarray             # Shape: (output_dim,)
    activation_type: ActivationType
    weight_gradients: np.ndarray  # Read-only, available after backward()
    bias_gradients: np.ndarray    # Read-only, available after backward()
```

Optimizer Classes

```
class Optimizer:  # Base class
    def update(self, layer: DenseLayer) -> None
    def reset(self) -> None
    learning_rate: float

class SGD(Optimizer):
    def __init__(self, learning_rate: float, momentum: float = 0.0)
    momentum: float

class Adam(Optimizer):
    def __init__(self, learning_rate: float = 0.001, beta1: float = 0.9,
                 beta2: float = 0.999, epsilon: float = 1e-8)
    beta1: float
    beta2: float
    epsilon: float

class RMSprop(Optimizer):
    def __init__(self, learning_rate: float = 0.001, decay_rate: float = 0.9,
                 epsilon: float = 1e-8)
    decay_rate: float
    epsilon: float
```

Loss Functions

All loss functions accept `y_pred` and `y_true` as numpy arrays and return a scalar loss value. Gradient functions return arrays with the same shape as the predictions.

```
# Regression losses
def mse(y_pred: np.ndarray, y_true: np.ndarray) -> float
def mse_gradient(y_pred: np.ndarray, y_true: np.ndarray) -> np.ndarray

def mae(y_pred: np.ndarray, y_true: np.ndarray) -> float
def mae_gradient(y_pred: np.ndarray, y_true: np.ndarray) -> np.ndarray

def huber(y_pred: np.ndarray, y_true: np.ndarray, delta: float = 1.0) -> float
def huber_gradient(y_pred: np.ndarray, y_true: np.ndarray, delta: float = 1.0) ->
np.ndarray

# Classification losses
def cross_entropy(y_pred: np.ndarray, y_true: np.ndarray, epsilon: float = 1e-15) ->
float
def cross_entropy_gradient(y_pred: np.ndarray, y_true: np.ndarray) -> np.ndarray

def binary_cross_entropy(y_pred: np.ndarray, y_true: np.ndarray, epsilon: float =
1e-15) -> float
def binary_cross_entropy_gradient(y_pred: np.ndarray, y_true: np.ndarray) ->
np.ndarray

# Generic interface
def compute_loss(y_pred: np.ndarray, y_true: np.ndarray, loss_type: LossType,
```

```
                delta: float = 1.0, epsilon: float = 1e-15) -> float
def compute_loss_gradient(y_pred: np.ndarray, y_true: np.ndarray, loss_type:
LossType,
                          delta: float = 1.0) -> np.ndarray
```

Activation Functions

```
def sigmoid(x: np.ndarray) -> np.ndarray
def tanh(x: np.ndarray) -> np.ndarray
def relu(x: np.ndarray) -> np.ndarray
def softmax(x: np.ndarray) -> np.ndarray

# Derivatives
def sigmoid_derivative(x: np.ndarray) -> np.ndarray
def tanh_derivative(x: np.ndarray) -> np.ndarray
def relu_derivative(x: np.ndarray) -> np.ndarray
def softmax_derivative(x: np.ndarray) -> np.ndarray

# Generic interface
def apply_activation(x: np.ndarray, activation_type: ActivationType) -> np.ndarray
def apply_activation_derivative(x: np.ndarray, activation_type: ActivationType) ->
np.ndarray
```

Enums

```
class ActivationType:
    RELU = ...
    SIGMOID = ...
    TANH = ...
    SOFTMAX = ...

class LossType:
    MSE = ...
    MAE = ...
    HUBER = ...
    CROSS_ENTROPY = ...
    BINARY_CROSS_ENTROPY = ...

class OptimizerType:
    SGD_TYPE = ...
    ADAM_TYPE = ...
    RMSPROP_TYPE = ...
```

Tips and Best Practices

1. Data Preprocessing

- Always use `dtype=np.float64` for numerical stability
- Normalize input features to have similar scales

- For classification, use one-hot encoding for multi-class problems

2. Network Architecture

- Start with smaller networks and increase size if needed
- Use ReLU activation for hidden layers in most cases
- Use sigmoid for binary classification output
- Use softmax for multi-class classification output
- Don’t use activation on regression output layers

3. Training

- Monitor loss during training to detect overfitting
- Use appropriate learning rates (0.01-0.001 is often good)
- Try different optimizers - Adam often works well out of the box
- Use mini-batch training for large datasets

4. Debugging

- Access gradients after `backward()` to check for vanishing/exploding gradients
- Verify that loss decreases during training
- Check that predictions make sense for your problem domain

This completes the DnnLib user manual. The library provides a solid foundation for neural network experimentation and learning, with the performance benefits of C++ and the ease of use of Python.