# Neural Network Framework Implementation

## A Step-by-Step Guide to Building Your Own Deep Learning Library

Morteza Maleki

Tarbiat Modares University

May 21, 2025

# Table of Contents

# Overview of the NPDL Framework

- A NumPy-based Deep Learning Framework
- Modular design with separate components:
  - Activation functions
  - Weight initializers
  - Neural network layers
  - Loss functions
  - Model construction
  - Optimizers
- Implementation follows object-oriented principles
- Designed for educational purposes

# Backpropagation - The Core of Neural Networks

- Backpropagation: The algorithm that powers neural network learning
- Key components of a neural network training loop:
  1. **Forward pass**: Compute predictions
  2. **Loss calculation**: Measure error
  3. **Backward pass**: Compute gradients
  4. **Parameter update**: Improve model
- Focus of this section: Understanding how gradients flow backward through the network
- Goal: Connect mathematical theory to our code implementation

# The Chain Rule - Foundation of Backpropagation

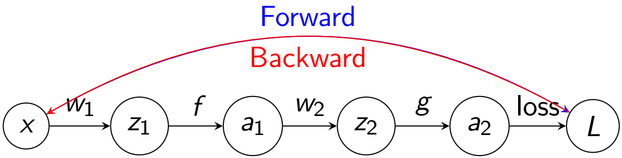
- Neural networks are composed of nested functions
- Chain rule from calculus: If $y = f(g(x))$, then $\frac{dy}{dx} = \frac{dy}{dg} \cdot \frac{dg}{dx}$
- For neural networks with many layers:

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial y_k} \cdot \frac{\partial y_k}{\partial a_j} \cdot \frac{\partial a_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ij}}$$

Where:

- $L$ is the loss
- $y_k$ is the output
- $a_j$ is the activation
- $z_j$ is the pre-activation
- $w_{ij}$ is a weight parameter

# Computational Graph Perspective




- Forward pass (blue): Compute outputs from inputs
- Backward pass (red): Propagate gradients from outputs to inputs
- Each node knows:
  - How to compute its output (forward)
  - How to compute gradients w.r.t. its inputs (backward)

# A 2-Layer Neural Network Example

Consider a simple 2-layer neural network:

$$z^{[1]} = W^{[1]}x + b^{[1]} \tag{1}$$

$$a^{[1]} = f(z^{[1]}) \tag{2}$$

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \tag{3}$$

$$\hat{y} = g(z^{[2]}) \tag{4}$$

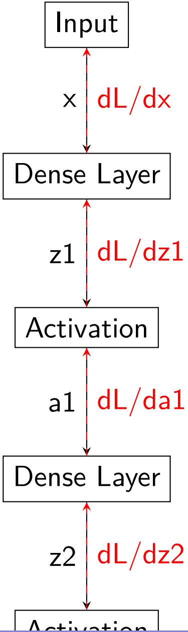
$$L = \text{loss}(\hat{y}, y) \tag{5}$$

- Computing gradients using the chain rule:

$$\frac{\partial L}{\partial W^{[2]}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial W^{[2]}} \tag{6}$$

$$\frac{\partial L}{\partial W^{[1]}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial a^{[1]}} \cdot \frac{\partial a^{[1]}}{\partial z^{[1]}} \cdot \frac{\partial z^{[1]}}{\partial W^{[1]}} \tag{7}$$

# Gradient Flow in Our Framework



# The Layer Interface in Our Framework

- Each layer in our framework follows the same interface:
  1. `forward()` - Computes layer output
  2. `backward()` - Computes gradients
  3. Stores parameters in `self.params`
  4. Stores gradients in `self.grads`
- This consistent interface enables automatic differentiation
- Models chain these layers together in both passes:
  - Forward: from first layer to last
  - Backward: from last layer to first

# Example: Gradient Flow in a Dense Layer

```python
class Dense(Layer):
    def forward(self, inputs):
        """Forward pass: y = Wx + b"""
        self.inputs = inputs  # Store for backward pass
        return np.dot(inputs, self.weights) + self.bias

    def backward(self, output_grad):
        """Backward pass: Compute gradients"""
        # Compute weight gradients (L /W)
        self.grads['weights'] = np.dot(self.inputs.T, output_grad)

        # Compute bias gradients (L /b)
        self.grads['bias'] = np.sum(output_grad, axis=0, keepdims=True)

        # Compute input gradients (L /x) to pass to previous layer
        return np.dot(output_grad, self.weights.T)

```

- Dense layer implements matrix multiplication: $y = Wx + b$
- Backward pass uses calculus rules:
    - $\frac{\partial L}{\partial W} = x^T \cdot \frac{\partial L}{\partial y}$
    - $\frac{\partial L}{\partial b} = \sum \frac{\partial L}{\partial y}$
    - $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot W^T$

# Example: Gradient Flow in an Activation Function

```python
class ReLU(Activation):
    def forward(self, x):
        """Forward pass: f(x) = max(0, x)"""
        self.input = x  # Store for backward pass
        self.output = np.maximum(0, x)
        return self.output

    def backward(self, output_grad):
        """Backward pass: Compute gradients"""
        # Derivative of ReLU: 1 if x > 0, else 0
        relu_grad = (self.input > 0).astype(float)

        # Chain rule: multiply upstream gradient with local gradient
        return output_grad * relu_grad

```

- Activation functions apply element-wise operations
- Backward pass computes local derivatives and applies chain rule
- For ReLU: $\frac{d}{dx}\max(0, x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$

# How Backward Passes Connect in the Framework

- Sequential model chains the backward passes automatically
- The flow of gradients:
  1. Loss function computes initial gradient: $\frac{\partial L}{\partial \hat{y}}$
  2. Each layer receives gradient w.r.t. its output: $\frac{\partial L}{\partial \text{out}}$
  3. Each layer computes:
     - Gradients for its parameters: $\frac{\partial L}{\partial \theta}$
     - Gradient w.r.t. its input: $\frac{\partial L}{\partial \text{in}}$
  4. Gradient w.r.t. input gets passed to previous layer
- Key insight: Each layer only needs to compute local gradients
- The framework handles chaining the gradients together

# Sequential Model's Backward Pass

```python
class Sequential(object):
    def __init__(self, layers=None):
        self.layers = layers if layers is not None else []

    def forward(self, inputs):
        """Forward pass through all layers in sequence"""
        for layer in self.layers:
            inputs = layer.forward(inputs)
        return inputs

    def backward(self, grad):
        """Backward pass through all layers in reverse"""
        for layer in reversed(self.layers):
        grad = layer.backward(grad)
        return grad

```

- `Sequential` model chains layers together
- **Forward**: Process inputs through layers in order
- **Backward**: Process gradients through layers in reverse order
- Each layer receives the gradient from the layer ahead
- The full chain rule is automatically implemented

# Implementation to Mathematics Mapping

| Component | Forward | Backward |
|---|---|---|
| Dense Layer | $z = Wx + b$ | $\frac{\partial L}{\partial W} = x^T \frac{\partial L}{\partial z}$<br>$\frac{\partial L}{\partial b} = \sum \frac{\partial L}{\partial z}$<br>$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} W^T$ |
| ReLU | $a = \max(0, z)$ | $\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a} \cdot \mathbf{1}_{z>0}$ |
| Sigmoid | $a = \frac{1}{1+e^{-z}}$ | $\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a} \cdot a(1-a)$ |
| Softmax | $a_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$ | Complex Jacobian matrix |
| MSE Loss | $L = \frac{1}{2}(y - \hat{y})^2$ | $\frac{\partial L}{\partial \hat{y}} = \hat{y} - y$ |

- Each component has a mathematical operation and its derivative
- The code implementation directly follows these equations
- Understanding the math makes the code implementation clear

# Updating Parameters with Optimizers

- After computing gradients, optimizers update parameters
- Stochastic Gradient Descent (SGD):

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta J(\theta)$$

- SGD with momentum:

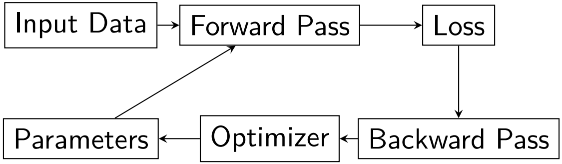
$$v_{t+1} = \gamma v_t + \alpha \nabla_\theta J(\theta) \quad (8)$$

$$\theta_{t+1} = \theta_t - v_{t+1} \quad (9)$$

- Each optimizer implements the `update(layer)` method:
  - Reads gradients from `layer.grads`
  - Updates parameters in `layer.params`
  - May maintain its own state (e.g., momentum)

# The Complete Training Loop

1. **Forward pass**: Input → Model → Prediction
2. **Loss calculation**: Compare prediction with target
3. **Backward pass**:
    - Start with loss gradient
    - Propagate through model in reverse
    - Collect parameter gradients
4. **Parameter update**:
    - Apply optimizer to update weights using gradients
5. Repeat for many examples and epochs



# Summary: Backpropagation in Our Framework

- Key components implementing backpropagation:
  - **Layers**: Compute outputs and gradients
  - **Activations**: Add non-linearity and corresponding gradients
  - **Loss functions**: Measure error and provide initial gradient
  - **Optimizers**: Update parameters using gradients
  - **Models**: Chain components together
- Design principles:
  - Modularity: Each component has specific responsibility
  - Unified interface: `forward()`/`backward()` methods
  - Mathematical correspondence: Code follows math directly
  - Automatic differentiation: Chain rule applied automatically
- Next, we'll explore each component in detail

# Activation Functions - Overview

- Activation functions introduce non-linearity
- Base abstract class with common interface
- Implementations of common activation functions:
    - Sigmoid
    - Tanh
    - ReLU
    - Softmax
- Each has forward and backward methods

# Activation Functions - Base Class

```python
class Activation(object):
    """Base class for all activation functions."""

    def forward(self, x):
        """Forward pass."""
        raise NotImplementedError

    def backward(self, output_grad):
        """Backward pass."""
        raise NotImplementedError
```

# Activation Functions - Sigmoid

```python
class Sigmoid(Activation):
    """Sigmoid activation function."""

    def forward(self, x):
        """Forward pass for sigmoid function.

        Args:
            x: Input numpy array.
        """
        self.output = 1.0 / (1.0 + np.exp(-x))
        return self.output

    def backward(self, output_grad):
        """Backward pass for sigmoid function.

        Args:
            output_grad: Gradient of the cost with respect to the
    output.
        """
        return output_grad * self.output * (1 - self.output)
```

# Sigmoid Activation Function - Mathematical Definition

- The sigmoid function is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Key properties:
  - Outputs values between 0 and 1
  - Smooth and differentiable everywhere
  - Historically popular but prone to vanishing gradient
  - S-shaped curve (sigmoid shape)
- Used primarily in:
  - Binary classification (output layer)
  - Early neural networks (mostly replaced by ReLU)
  - Gates in recurrent neural networks (LSTM, GRU)

# Sigmoid Activation Function - Gradient Derivation

$$
\begin{aligned}
\sigma(x) &= \frac{1}{1+e^{-x}} && (10)\\
\frac{d\sigma(x)}{dx} &= \frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right) && (11)\\
&= \frac{d}{dx}(1+e^{-x})^{-1} && (12)\\
&= -(1+e^{-x})^{-2}\cdot\frac{d}{dx}(1+e^{-x}) && (13)\\
&= -(1+e^{-x})^{-2}\cdot(-e^{-x}) && (14)\\
&= \frac{e^{-x}}{(1+e^{-x})^2} && (15)\\
&= \frac{1}{1+e^{-x}}\cdot\frac{e^{-x}}{1+e^{-x}} && (16)\\
&= \frac{1}{1+e^{-x}}\cdot\left(1-\frac{1}{1+e^{-x}}\right) && (17)
\end{aligned}
$$

# Sigmoid Activation Function - Implementation Details

- Forward pass:
  - Calculate $\sigma(x) = \frac{1}{1+e^{-x}}$
  - Store output for use in backward pass
  - Handle numerical stability with clipping for extreme values
- Backward pass:
  - Use the elegant form of gradient: $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$
  - Multiply input gradient by this derivative (chain rule)
  - No need to reference original input - only output is needed
- Implementation challenges:
  - Saturation for large positive/negative inputs
  - Vanishing gradient problem when chaining multiple sigmoids

# ReLU Activation Function - Detailed Overview

- Rectified Linear Unit defined as:

$$\mathrm{ReLU}(x) = \max(0, x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

- Key properties:
  - Simple, computationally efficient
  - Non-linear despite simple form
  - Sparse activation (many neurons output zero)
  - No vanishing gradient for positive inputs
  - Allows for deeper networks
- Limitations:
  - "Dying ReLU" problem when neurons get stuck at 0
  - Non-zero centered outputs
  - Unbounded positive activation

# ReLU Activation Function - Gradient Analysis

$$\frac{d\text{ReLU}(x)}{dx} = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \\ \text{undefined} & \text{if } x = 0 \end{cases} \tag{19}$$

- Gradient is either 0 or 1 (easy to compute)
- No saturation for positive values (solves vanishing gradient)
- Promotes sparsity in the network
- Gradient at $x = 0$ is technically undefined, but usually set to 0 or 1
- Computationally efficient gradient - just check if input was positive

# ReLU Variants - LeakyReLU, PReLU, ELU

- **LeakyReLU**: Allows small negative values with a fixed slope

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$

  where $\alpha$ is a small constant (e.g., 0.01)
- **Parametric ReLU (PReLU)**: Learns the slope parameter $\alpha$ during training
- **Exponential Linear Unit (ELU)**:

$$\text{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$

- These variants help address the "dying ReLU" problem while maintaining ReLU's advantages

# Activation Functions - ReLU

```python
class ReLU(Activation):
    """ReLU activation function."""

    def forward(self, x):
        """Forward pass for ReLU function.

        Args:
            x: Input numpy array.
        """
        self.input = x
        self.output = np.maximum(x, 0)
        return self.output

    def backward(self, output_grad):
        """Backward pass for ReLU function.

        Args:
            output_grad: Gradient of the cost with respect to the
    output.
        """
        return output_grad * (self.input > 0)
```

# Softmax Activation Function - Comprehensive Overview

- Softmax converts a vector of values into a probability distribution:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

- Key properties:
  - Outputs are in range [0,1] and sum to 1
  - Preserves relative order of inputs (monotonic)
  - Emphasizes largest values, suppresses smaller ones
  - Not element-wise (depends on all input values)
- Use cases:
  - Output layer for multi-class classification
  - Attention mechanisms in transformers
  - Any scenario requiring normalized probabilities

# Softmax Activation Function - Numerical Stability

- Naive implementation can cause numerical overflow:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

- Stabilized implementation:

$$\text{softmax}(x_i) = \frac{e^{x_i - \max(x)}}{\sum_{j=1}^{n} e^{x_j - \max(x)}}$$

- Subtracting the maximum value prevents overflow:
  - Doesn't change the output (same relative proportions)
  - Ensures at least one exponent equals 1 (when $x_i = \max(x)$)
  - Makes largest exponent manageable
  - Critical for stable computation with larger numbers

# Softmax Activation Function - Gradient

- The gradient is a Jacobian matrix:

$$\frac{\partial \text{softmax}(x_i)}{\partial x_j} = \begin{cases} \text{softmax}(x_i)(1 - \text{softmax}(x_i)) & \text{if } i = j \\ -\text{softmax}(x_i)\text{softmax}(x_j) & \text{if } i \neq j \end{cases}$$

- In matrix form:

$$\nabla_{\mathbf{x}} \text{softmax}(\mathbf{x}) = \text{diag}(\text{softmax}(\mathbf{x})) - \text{softmax}(\mathbf{x}) \otimes \text{softmax}(\mathbf{x})$$

- In practice, often combined with cross-entropy loss for simplification
- When used with cross-entropy, gradient simplifies to: $(\hat{y} - y)$

# Activation Functions - Softmax

```
class Softmax(Activation):
    """Softmax activation function."""

    def forward(self, x):
        """Forward pass for Softmax function.

        Args:
            x: Input numpy array.
        """
        exp_values = np.exp(x - np.max(x, axis=1, keepdims=True))
        self.output = exp_values / np.sum(exp_values,
                                           axis=1, keepdims=True)
        return self.output

    def backward(self, output_grad):
        """Backward pass for Softmax function.

        Args:
            output_grad: Gradient of the cost with respect to the
    output.
        """
        # Simplified backward pass
        return output_grad
```

# Neural Network Layers - Overview

- Building blocks of neural networks
- Base Layer interface
- Implementations:
  - Dense (Fully connected)
  - Dropout (Regularization)
  - Flatten (Reshaping)
- Each layer implements forward and backward passes

# Layers - Base Class

```python
class Layer(object):
    """Base class for all layers."""

    def __init__(self):
        """Initialize the layer."""
        self.params = {}
        self.grads = {}

    def forward(self, inputs):
        """Forward pass.

        Args:
            inputs: Input data.
        """
        raise NotImplementedError

    def backward(self, output_grad):
        """Backward pass.

        Args:
            output_grad: Gradient of the cost with respect to the output.
        """
        raise NotImplementedError
```

# Layers - Dense Layer (1/2)

```python
class Dense(Layer):
    """Fully connected layer."""

    def __init__(self, n_units, input_shape=None,
                 weight_initializer=None,
                 bias_initializer=None):
        """Initialize the dense layer.

        Args:
            n_units: Number of output units.
            input_shape: Shape of the input data.
            weight_initializer: Weight initializer.
            bias_initializer: Bias initializer.
        """
        super(Dense, self).__init__()
        self.n_units = n_units
        self.input_shape = input_shape

        self.weight_initializer = weight_initializer
        if self.weight_initializer is None:
            self.weight_initializer = HeNormal()

        self.bias_initializer = bias_initializer
        if self.bias_initializer is None:
            self.bias_initializer = Zero()
```

# Layers - Dense Layer (2/2)

```
    def forward(self, inputs):
        """Forward pass for dense layer.

        Args:
            inputs: Input data.
        """
        self.inputs = inputs

        if not hasattr(self, 'weights'):
            self.weights = self.weight_initializer(
                (self.input_shape, self.n_units))
            self.bias = self.bias_initializer((1, self.n_units))

            self.params['weights'] = self.weights
            self.params['bias'] = self.bias

        return np.dot(self.inputs, self.weights) + self.bias

    def backward(self, output_grad):
        """Backward pass for dense layer.

        Args:
            output_grad: Gradient of the cost with respect to the
    output.
        """
```

# Layers - Dropout Layer

```python
class Dropout(Layer):
    """Dropout layer."""

    def __init__(self, dropout_rate):
        """Initialize the dropout layer.

        Args:
            dropout_rate: Dropout rate.
        """
        super(Dropout, self).__init__()
        self.dropout_rate = dropout_rate

    def forward(self, inputs, training=True):
        """Forward pass for dropout layer.

        Args:
            inputs: Input data.
            training: Whether in training mode.
        """
        self.inputs = inputs

        if training:
            self.mask = np.random.binomial(
                1, 1 - self.dropout_rate, size=inputs.shape) / (1 -
    self.dropout_rate)
```

# Loss Functions - Overview

- Measure model performance
- Base Loss abstract class
- Common loss functions:
  - Mean Squared Error (MSE)
  - Categorical Cross-Entropy
  - Binary Cross-Entropy
- Each implements forward and gradient calculations

# Loss Functions - Base Class

```python
class Loss(object):
    """Base class for all loss functions."""

    def forward(self, y_true, y_pred):
        """Forward pass.

        Args:
            y_true: Ground truth values.
            y_pred: Predicted values.
        """
        raise NotImplementedError

    def gradient(self, y_true, y_pred):
        """Gradient of the loss function.

        Args:
            y_true: Ground truth values.
            y_pred: Predicted values.
        """
        raise NotImplementedError
```

# Loss Functions - MSE

```python
class MeanSquaredError(Loss):
    """Mean squared error loss function."""

    def forward(self, y_true, y_pred):
        """Forward pass for mean squared error.

        Args:
            y_true: Ground truth values.
            y_pred: Predicted values.
        """
        return 0.5 * np.power(y_pred - y_true, 2).mean()

    def gradient(self, y_true, y_pred):
        """Gradient of the mean squared error.

        Args:
            y_true: Ground truth values.
            y_pred: Predicted values.
        """
        return y_pred - y_true
```

# Loss Functions - Categorical Cross-Entropy

```python
class CategoricalCrossentropy(Loss):
    """Categorical cross-entropy loss function."""

    def forward(self, y_true, y_pred):
        """Forward pass for categorical cross-entropy.

        Args:
            y_true: Ground truth values.
            y_pred: Predicted values.
        """
        # Clip to avoid log(0)
        y_pred = np.clip(y_pred, 1e-15, 1 - 1e-15)
        return -np.sum(y_true * np.log(y_pred)) / y_true.shape[0]

    def gradient(self, y_true, y_pred):
        """Gradient of the categorical cross-entropy.

        Args:
            y_true: Ground truth values.
            y_pred: Predicted values.
        """
        # Clip to avoid division by zero
        y_pred = np.clip(y_pred, 1e-15, 1 - 1e-15)
        return -y_true / y_pred / y_true.shape[0]
```

# Models - Overview

- Sequential model for chaining layers
- Methods for training and evaluation
- Forward/backward pass implementation
- Training loop with batch processing
- Model evaluation and prediction

# Models - Sequential (1/2)

```python
class Sequential(object):
    """Sequential model."""

    def __init__(self, layers=None):
        """Initialize the model.

        Args:
            layers: List of layers.
        """
        self.layers = layers if layers is not None else []

    def add(self, layer):
        """Add a layer to the model.

        Args:
            layer: Layer to add.
        """
        self.layers.append(layer)
```

# Models - Sequential (2/2)

```
    def forward(self, inputs, training=True):
        """Forward pass.

        Args:
            inputs: Input data.
            training: Whether in training mode.
        """
        for layer in self.layers:
            if hasattr(layer, 'training'):
                inputs = layer.forward(inputs, training)
            else:
                inputs = layer.forward(inputs)
        return inputs

    def backward(self, grad):
        """Backward pass.

        Args:
            grad: Gradient of the cost with respect to the output.
        """
        for layer in reversed(self.layers):
            grad = layer.backward(grad)
        return grad
```

# Models - Training

```python
def fit(self, x, y, epochs=100, batch_size=32,
        loss_fn=None, optimizer=None,
        validation_data=None, verbose=True):
    """Train the model.

    Args:
        x: Input data.
        y: Target data.
        epochs: Number of epochs.
        batch_size: Batch size.
        loss_fn: Loss function.
        optimizer: Optimizer.
        validation_data: Validation data.
        verbose: Whether to print progress.
    """
    if loss_fn is None:
        loss_fn = MeanSquaredError()

    if optimizer is None:
        optimizer = SGD()

    # Training loop implementation
    for epoch in range(epochs):
        # Process mini-batches
        # Update weights using optimizer
```

# Optimizers - Overview

- Update model parameters based on gradients
- Base Optimizer abstract class
- Common optimization algorithms:
  - SGD (Stochastic Gradient Descent)
  - Adam (Adaptive Moment Estimation)
  - RMSprop
- Each implements the update method

# Optimizers - Base Class

```python
class Optimizer(object):
    """Base class for all optimizers."""

    def __init__(self, learning_rate=0.01):
        """Initialize the optimizer.

        Args:
            learning_rate: Learning rate.
        """
        self.learning_rate = learning_rate

    def update(self, layer):
        """Update the layer weights.

        Args:
            layer: Layer to update.
        """
        raise NotImplementedError
```

# Optimizers - SGD

```python
class SGD(Optimizer):
    """Stochastic gradient descent optimizer."""

    def __init__(self, learning_rate=0.01, momentum=0.0):
        """Initialize the SGD optimizer.

        Args:
            learning_rate: Learning rate.
            momentum: Momentum factor.
        """
        super(SGD, self).__init__(learning_rate)
        self.momentum = momentum
        self.velocity = {}

    def update(self, layer):
        """Update the layer weights.

        Args:
            layer: Layer to update.
        """
        for param_name in layer.params:
            # Initialize velocity for the parameter if not exists
            if param_name not in self.velocity:
                self.velocity[param_name] = np.zeros_like(
                    layer.params[param_name])
```

# Optimizers - Adam

```python
class Adam(Optimizer):
    """Adam optimizer."""

    def __init__(self, learning_rate=0.001, beta_1=0.9,
                 beta_2=0.999, epsilon=1e-8):
        """Initialize the Adam optimizer.

        Args:
            learning_rate: Learning rate.
            beta_1: Exponential decay rate for first moment.
            beta_2: Exponential decay rate for second moment.
            epsilon: Small constant for numerical stability.
        """
        super(Adam, self).__init__(learning_rate)
        self.beta_1 = beta_1
        self.beta_2 = beta_2
        self.epsilon = epsilon
        self.m = {}  # First moment
        self.v = {}  # Second moment
        self.t = 0   # Timestep

    def update(self, layer):
        """Update implementation with moment calculations"""
```

# Building a Complete Neural Network

```python
# Import all components
from npdl.models import Sequential
from npdl.layers import Dense, Dropout
from npdl.activations import ReLU, Softmax
from npdl.initializers import HeNormal, Zero
from npdl.losses import CategoricalCrossentropy
from npdl.optimizers import Adam

# Create a model
model = Sequential()
model.add(Dense(128, input_shape=784,
               weight_initializer=HeNormal(),
               bias_initializer=Zero()))
model.add(ReLU())
model.add(Dropout(0.2))
model.add(Dense(64))
model.add(ReLU())
model.add(Dropout(0.2))
model.add(Dense(10))
model.add(Softmax())

# Compile and train
model.fit(x_train, y_train, epochs=10, batch_size=32,
         loss_fn=CategoricalCrossentropy(),
         optimizer=Adam(learning_rate=0.001),
```

# Summary and Next Steps

- We've covered the complete implementation of:
  - Activation functions
  - Weight initializers
  - Neural network layers
  - Loss functions
  - Model construction
  - Optimizers
- Possible extensions:
  - Convolutional layers
  - Recurrent layers
  - Batch normalization
  - More advanced optimizers
- Practical exercises to implement and test