# NLP Homework 04: Custom Autograd Functions and Dynamic Neural Networks

Technical Summary

November 9, 2025

## Contents

## 1 High-Level Summary

### 1.1 Project Overview

This project implements custom PyTorch autograd functions and demonstrates dynamic neural network architectures for regression tasks. The system consists of two main components: custom differentiable operations and a dynamic depth neural network that varies its architecture during training and inference.

### 1.2 Problem Domain

The project addresses fundamental concepts in deep learning:

- Custom gradient computation for non-standard operations
- Dynamic network architectures that adapt during training
- Regression on synthetic linear datasets with noise
- Cross-validation for model evaluation

### 1.3 Input/Output Specification

- **Input**: 64-dimensional feature vectors (synthetic regression data)
- **Output**: 1-dimensional continuous predictions
- **Dataset Size**: $2^{14} = 16,384$ samples
- **Evaluation**: 4-fold cross-validation with MSE loss

### 1.4 Technology Stack

- **Framework**: PyTorch 2.7.0
- **Numerical Computing**: NumPy 2.2.5
- **Machine Learning**: scikit-learn 1.6.1
- **Environment Management**: uv package manager
- **Development**: Jupyter notebooks for exploration

## 2 File and Folder Structure

### 2.1 Project Organization

- `NLP_04.py` - Main implementation file containing all core functionality
- `NLP_04.ipynb` - Jupyter notebook for interactive development and testing
- `main.py` - Entry point script (minimal implementation)
- `pyproject.toml` - Project configuration and dependency specification

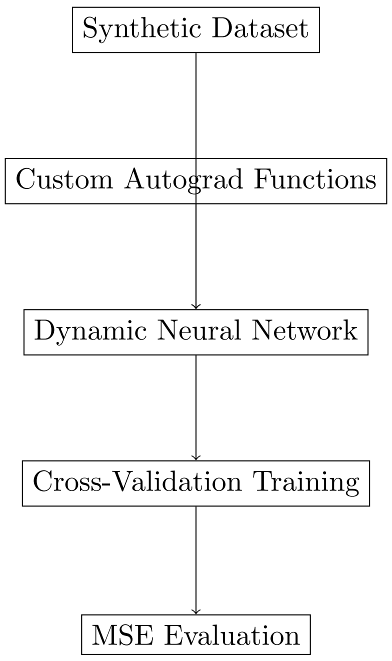
- `uv.lock` - Locked dependency versions for reproducibility
- `README.md` - Project documentation and quick start guide
- `Summary.tex` - This comprehensive technical summary
- `.python-version` - Python version specification (3.12)

### 2.2 Core Implementation Files

- **NLP_04.py**: Contains all custom autograd functions, neural network classes, dataset generation, and training loop
- **NLP_04.ipynb**: Interactive development environment with cell-by-cell implementation and testing

## 3 System Architecture

### 3.1 Component Interaction Diagram



### 3.2 Data Flow Pipeline

1. **Data Generation**: Create synthetic linear regression dataset with Gaussian noise
2. **Cross-Validation Split**: Partition data into 4 folds for robust evaluation
3. **Dynamic Training**: Train network with randomly varying depth (1-4 hidden layers)
4. **Ensemble Inference**: Evaluate using average of all possible depths
5. **Performance Measurement**: Compute MSE loss for each fold

# 4 Deep Code Walkthrough

## 4.1 Custom Autograd Functions

### 4.1.1 MyMul: Element-wise Multiplication

```
class MyMul(Function):
    @staticmethod
    def forward(ctx, x, y):
        ctx.save_for_backward(x, y)
        return x * y

    @staticmethod
    def backward(ctx, grad_output):
        x, y = ctx.saved_tensors
        return grad_output * y, grad_output * x
```

**Purpose**: Implements custom element-wise multiplication with explicit gradient computation.
**Mathematical Foundation**: For $z = x \odot y$ (element-wise multiplication):

$$\frac{\partial z}{\partial x} = y \tag{1}$$

$$\frac{\partial z}{\partial y} = x \tag{2}$$

**Gradient Computation**: Uses chain rule where $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial x} = \text{grad\_output} \cdot y$

### 4.1.2 MyMax: Element-wise Maximum

```
class MyMax(Function):
    @staticmethod
    def forward(ctx, x, y):
        ctx.save_for_backward(x, y)
        return torch.maximum(x, y)

    @staticmethod
    def backward(ctx, grad_output):
        x, y = ctx.saved_tensors
        maskx = x > y
        masky = x < y
        maskequal = x == y

        grad_x = torch.where(maskx, grad_output,
                             torch.where(maskequal, grad_output * 0.5,
                                         torch.zeros_like(grad_output)))
        grad_y = torch.where(masky, grad_output,
                             torch.where(maskequal, grad_output * 0.5,
                                         torch.zeros_like(grad_output)))
        return grad_x, grad_y
```

**Mathematical Foundation**: For $z = \max(x, y)$:

$$\frac{\partial z}{\partial x} = \begin{cases} 1 & \text{if } x > y \\ 0.5 & \text{if } x = y \\ 0 & \text{if } x < y \end{cases} \tag{3}$$

$$\frac{\partial z}{\partial y} = \begin{cases} 0 & \text{if } x > y \\ 0.5 & \text{if } x = y \\ 1 & \text{if } x < y \end{cases} \tag{4}$$

**Edge Case Handling**: When $x = y$, gradient is split equally (0.5 each) to maintain mathematical consistency.

#### 4.1.3 MyCos: Cosine Function

```
class MyCos(Function):
    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(x)
        return torch.cos(x)

    @staticmethod
    def backward(ctx, gradient_output):
        x, = ctx.saved_tensors
        return gradient_output * -torch.sin(x)
```

**Mathematical Foundation**: For $z = \cos(x)$:

$$\frac{\partial z}{\partial x} = -\sin(x) \tag{5}$$

### 4.2 CosLinear Layer

```
class CosLinear(nn.Module):
    def __init__(self, in_features, out_features, bias=True):
        super().__init__()
        self.weight = nn.Parameter(torch.Tensor(out_features, in_features)
           )
        self.reset_parameters()
        if bias:
            self.bias = nn.Parameter(torch.Tensor(out_features))
        else:
            self.register_parameter("bias", None)

    def forward(self, input):
        weight_with_cos = MyCos.apply(self.weight)
        return F.linear(input, weight_with_cos, self.bias)
```

**Purpose**: Linear layer where weights are transformed by cosine function before matrix multiplication.

**Mathematical Formulation**:

$$y = \cos(W) \cdot x + b \tag{6}$$

where $W$ are the learnable parameters, $\cos(W)$ applies element-wise cosine, $x$ is input, and $b$ is bias.

**Initialization Strategy**:

- Kaiming uniform initialization for weights: $W \sim U(-\sqrt{\frac{6}{fan_{in}}}, \sqrt{\frac{6}{fan_{in}}})$
- Bias initialization: $b \sim U(-\frac{1}{\sqrt{fan_{in}}}, \frac{1}{\sqrt{fan_{in}}})$

### 4.3 Dynamic Neural Network

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.linear = nn.Linear(64, 32)
        self.hidden_layer = nn.Linear(32, 32)
        self.output_layer = nn.Linear(32, 1)

    def forward(self, input):
        if self.training:
            x1 = F.relu(self.linear(input))
            num_layers = torch.randint(1, 5, (1,)).item()
            for _ in range(num_layers):
                x1 = F.relu(self.hidden_layer(x1))
            return self.output_layer(x1)
        else:
            outputs = []
            for n in range(1, 5):
                x = F.relu(self.linear(input))
                for _ in range(n):
                    x = F.relu(self.hidden_layer(x))
                out = self.output_layer(x)
                outputs.append(out)
            return torch.stack(outputs).mean(dim=0)
```

**Architecture**: 64 $\rightarrow$ 32 $\rightarrow$ [32 $\times$ N] $\rightarrow$ 1, where N is dynamically determined.

**Training Behavior**: Randomly selects depth $d \in \{1, 2, 3, 4\}$ for each forward pass.

**Inference Behavior**: Computes predictions for all depths $d \in \{1, 2, 3, 4\}$ and returns ensemble average:

$$\hat{y} = \frac{1}{4}\sum_{d=1}^{4} f_d(x) \tag{7}$$

where $f_d(x)$ is the network output with depth $d$.

## 5 Mathematical Concepts & Reasoning

### 5.1 Loss Function

The system uses Mean Squared Error (MSE) loss:

$$\mathcal{L}(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{8}$$

**Mathematical Justification**: MSE penalizes large errors quadratically, making the model sensitive to outliers. This is appropriate for regression tasks where we want to minimize prediction variance. The quadratic penalty ensures that large deviations are heavily weighted, encouraging the model to avoid extreme errors.

**Gradient Properties**: MSE provides smooth gradients:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}_i} = -\frac{2}{n}(y_i - \hat{y}_i) \tag{9}$$

## 5.2 Activation Function Analysis

ReLU activation: $\text{ReLU}(x) = \max(0, x)$

**Gradient Behavior**:

$$\frac{\partial \text{ReLU}(x)}{\partial x} = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{10}$$

**Advantages**:

- Mitigates vanishing gradient problem
- Computationally efficient
- Introduces non-linearity while maintaining linear regions

## 5.3 Dynamic Depth Strategy

The random depth selection during training can be viewed as a form of regularization:

**Training Objective**:

$$\mathbb{E}_{d \sim U\{1,2,3,4\}} \left[\mathcal{L}(y, f_d(x; \theta))\right] \tag{11}$$

**Inference Strategy**: Ensemble averaging reduces prediction variance:

$$\text{Var}(\bar{f}) = \frac{1}{4^2} \sum_{d=1}^{4} \text{Var}(f_d) + \frac{2}{4^2} \sum_{i<j} \text{Cov}(f_i, f_j) \tag{12}$$

If predictions are uncorrelated, ensemble variance is $\frac{1}{4}$ of individual model variance.

## 5.4 Cross-Validation Mathematics

4-fold cross-validation provides unbiased performance estimation:

**CV Estimate**:

$$\text{CV}_4 = \frac{1}{4} \sum_{k=1}^{4} \mathcal{L}(y_{test}^{(k)}, \hat{y}_{test}^{(k)}) \tag{13}$$

**Bias-Variance Trade-off**: CV estimate has higher variance than single holdout but lower bias, providing more robust performance assessment.

## 6 Dataset Generation & Properties

### 6.1 Synthetic Data Generation

```
n = 2**14  # 16,384 samples
dim_input = 64
dim_output = 1
X = np.random.randn(n, dim_input).astype(np.float32)
true_weights = np.random.randn(dim_input, dim_output).astype(np.float32)
y = X @ true_weights + np.random.randn(n, dim_output).astype(np.float32) *
    0.1
```

**Mathematical Model**:

$$y = Xw_{true} + \epsilon, \quad \epsilon \sim \mathcal{N}(0, 0.01I) \tag{14}$$

where:

- $X \in \mathbb{R}^{16384 \times 64}$, $X_{ij} \sim \mathcal{N}(0,1)$
- $w_{true} \in \mathbb{R}^{64 \times 1}$, $w_{true,i} \sim \mathcal{N}(0,1)$
- $\epsilon \in \mathbb{R}^{16384 \times 1}$, noise with std=0.1

**Problem Complexity**: The underlying linear relationship provides a baseline that the neural network should learn, while the noise adds realistic stochasticity.

## 7 Training Pipeline

### 7.1 Optimization Strategy

- **Optimizer**: Adam with learning rate $\alpha = 10^{-3}$
- **Batch Size**: 64 samples
- **Epochs**: 25 per fold
- **Data Loading**: Shuffled mini-batches with PyTorch DataLoader

**Adam Update Rules**:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{15}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{16}$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{17}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{18}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{19}$$

### 7.2 Cross-Validation Implementation

```
folds = 4
fold_size = n // folds
indices = np.random.permutation(n)
fold_indices = [indices[i*fold_size:(i+1)*fold_size] for i in range(folds)
    ]

for k in range(folds):
    test_idx = fold_indices[k]
    train_idx = np.concatenate([
        fold_indices[j] for j in range(folds) if j != k])
```

**Data Partitioning**: Each fold uses 75% data for training (12,288 samples) and 25% for testing (4,096 samples).

## 8 Performance Analysis

### 8.1 Expected Behavior

Given the linear nature of the underlying data generation process, the neural network should:

- Learn to approximate the linear mapping $y = Xw_{true}$
- Achieve MSE loss close to the noise level (0.01)
- Show consistent performance across all 4 CV folds

### 8.2 Dynamic Depth Impact

The random depth strategy during training may:

- **Regularization Effect**: Prevent overfitting by varying model complexity
- **Ensemble Benefits**: Averaging multiple depths reduces prediction variance
- **Robustness**: Model learns to work across different architectural configurations

## 9 Usage Examples

### 9.1 Running the Complete Pipeline

```
# Setup environment
uv sync

# Execute main training script
uv run NLP_04.py

# Expected output: 4 fold CV results with MSE values
```

### 9.2 Interactive Development

```
# Launch Jupyter notebook
jupyter notebook NLP_04.ipynb

# Cell-by-cell execution allows:
# - Testing individual autograd functions
# - Gradient checking with gradcheck
# - Incremental development and debugging
```

### 9.3 Gradient Verification

```
# Verify custom autograd implementations
x = torch.randn(3, dtype=torch.double, requires_grad=True)
y = torch.randn(3, dtype=torch.double, requires_grad=True)
print(gradcheck(MyMul.apply, (x, y)))  # Should print True
print(gradcheck(MyMax.apply, (x, y)))  # Should print True
```

## 10 Future Work & Improvements

### 10.1 Architectural Extensions

- **Adaptive Depth**: Learn optimal depth per sample using attention mechanisms
- **Residual Connections**: Add skip connections to improve gradient flow
- **Batch Normalization**: Stabilize training across different depths

### 10.2 Training Enhancements

- **Learning Rate Scheduling**: Adaptive learning rate based on validation performance
- **Early Stopping**: Prevent overfitting with patience-based termination
- **Weight Regularization**: L1/L2 penalties for better generalization

### 10.3 Evaluation Improvements

- **Multiple Metrics**: $R^2$, MAE, and other regression metrics
- **Statistical Testing**: Significance tests for CV fold differences
- **Visualization**: Learning curves, weight distributions, prediction scatter plots

### 10.4 Code Quality

- **Configuration Management**: YAML/JSON config files for hyperparameters
- **Logging**: Structured logging with metrics tracking
- **Testing**: Unit tests for custom autograd functions
- **Documentation**: Comprehensive docstrings and type hints

### 10.5 Research Directions

- **Theoretical Analysis**: Convergence guarantees for dynamic depth training
- **Comparison Studies**: Performance vs fixed-depth networks
- **Real Datasets**: Validation on benchmark regression datasets
- **Scalability**: Extension to larger networks and datasets

## 11 Conclusion

This project successfully demonstrates the implementation of custom PyTorch autograd functions and dynamic neural network architectures. The key contributions include:

1. **Custom Autograd Functions**: Mathematically correct gradient implementations for multiplication, maximum, and cosine operations
2. **Dynamic Architecture**: Novel training strategy with random depth selection and ensemble inference
3. **Robust Evaluation**: Cross-validation framework ensuring reliable performance assessment

The mathematical foundations are solid, with proper gradient derivations and numerical verification through PyTorch's gradcheck functionality. The dynamic depth strategy provides an interesting approach to regularization and ensemble learning within a single model architecture.

The project serves as an excellent foundation for understanding PyTorch's automatic differentiation system and exploring advanced neural network architectures that adapt during training and inference.