Neural Networks and Deep Learning
Lecture Notes

Based on Andrej Karpathy's Lectures

Andrej Karpathy
OpenAI / Tesla

August 19, 2025

These notes are based on Andrej Karpathy's lectures on neural networks and deep learning fundamentals.

"The goal is to build a strong foundation in neural networks by implementing everything from scratch and understanding the math."

Contents

1 Lecture 1: The Spelled-Out Intro to Neural Networks and Backpropagation with Micrograd

Abstract

This lecture provides a thorough deep dive into the foundational concepts of neural network training, focusing on automatic differentiation (autograd) and backpropagation. Using a simplified Python library called Micrograd, we will build a neural network from scratch, demystifying the "under the hood" mechanisms. The pedagogical approach emphasizes understanding scalar operations and the chain rule, avoiding the complexities of high-dimensional tensors initially to promote intuitive grasp. We will cover derivatives, the Value object, manual and automatic backpropagation, the training loop, and compare Micrograd's approach to production-grade libraries like PyTorch.

1.1 Introduction to Neural Network Training and Micrograd

Neural network training is fundamentally about iteratively tuning the weights of a neural network to minimize a loss function, thereby improving the network's accuracy. This process relies heavily on an algorithm called backpropagation, which efficiently calculates the gradient of the loss function with respect to the network's weights.

1.1.1 Micrograd: A Pedagogical Autograd Engine

Micrograd is a Python library designed to illustrate the core principles of automatic gradient computation (autograd) and backpropagation. Its primary goal is pedagogical:

- It operates on scalar values (single numbers) rather than complex N-dimensional tensors commonly found in modern deep learning libraries like PyTorch or JAX. This simplification allows for a clearer understanding of the underlying mathematical operations and the chain rule without being bogged down by tensor dimensionality.
- It is intentionally concise, with the core autograd engine (responsible for backpropagation) implemented in roughly 100 lines of very simple Python code. The neural network library built on top of it (nn.py) is also quite minimal, defining basic structures like neurons, layers, and multi-layer perceptrons (MLPs).

While Micrograd is not for production use due to its scalar-based nature and lack of parallelization, it provides a crucial foundation for understanding how modern deep learning frameworks function at their mathematical core.

1.2 Understanding Derivatives: The Intuition

Before diving into backpropagation, it's essential to have a strong intuitive understanding of what a derivative represents.

1.2.1 Derivative of a Single-Variable Function

A derivative measures the sensitivity of a function's output to a tiny change in its input. It tells us the slope of the function at a specific point, indicating whether the function is increasing or decreasing and by how much.

Consider a scalar-valued function $f(x) = 3x^2 - 4x + 5$. We can numerically approximate the derivative at a point x using the definition:

$$\frac{df}{dx} \approx \frac{f(x+h) - f(x)}{h}$$

where h is a very small number (e.g., 0.001).

```python
import math
import numpy as np
import matplotlib.pyplot as plt

# Define the function
def f(x):
    return 3*x**2 - 4*x + 5

# Example: calculate f(3.0)
print(f(3.0))  # Output: 20.0

# Plotting the function
xs = np.arange(-5, 5, 0.25)
ys = f(xs)
plt.plot(xs, ys)
plt.title("Function f(x) = 3x^2 - 4x + 5")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid(True)
plt.show()

# Numerical derivative at x = 3.0
h = 0.001
x = 3.0
f_x = f(x)
f_x_plus_h = f(x + h)
slope_at_3 = (f_x_plus_h - f_x) / h
print(f"Slope at x={x}: {slope_at_3}")  # Expected: ~14.0 (Analytical: 6x - 4 -> 6*3 - 4 = 14)

# Numerical derivative at x = -3.0
x_neg = -3.0
f_x_neg = f(x_neg)
f_x_plus_h_neg = f(x_neg + h)
slope_at_neg_3 = (f_x_plus_h_neg - f_x_neg) / h
print(f"Slope at x={x_neg}: {slope_at_neg_3}")  # Expected: ~-22.0 (Analytical: 6*(-3) - 4 = -22)
```

Listing 1: Numerical Derivative Calculation

The sign of the derivative indicates the direction of change:

- Positive derivative: Function increases if input is slightly increased.
- Negative derivative: Function decreases if input is slightly increased.
- Zero derivative: Function is momentarily flat (e.g., at a minimum or maximum).

1.2.2 Derivative with Multiple Inputs (Partial Derivatives)

When a function has multiple inputs, we talk about partial derivatives. A partial derivative with respect to one input tells us how the output changes when only that specific input is slightly nudged, while all other inputs are held constant.

Consider the expression $d = a \times b + c$ where $a = 2$, $b = -3$, $c = 10$. The output $d = 4$. Let's find the partial derivative of d with respect to a, i.e., $\frac{\partial d}{\partial a}$:

```python
# Define the multi-input function
def func_d(a, b, c):
    return a * b + c

# Initial values
a, b, c = 2.0, -3.0, 10.0
d1 = func_d(a, b, c)  # d1 = 4.0

h = 0.001

# d(d)/d(a)
a_bumped = a + h
d2_a = func_d(a_bumped, b, c)
slope_a = (d2_a - d1) / h
print(f"d(d)/d(a): {slope_a}")  # Expected: -3.0 (which is the
    value of b)

# d(d)/d(b)
b_bumped = b + h
d2_b = func_d(a, b_bumped, c)
slope_b = (d2_b - d1) / h
print(f"d(d)/d(b): {slope_b}")  # Expected: 2.0 (which is the
   value of a)

# d(d)/d(c)
c_bumped = c + h
d2_c = func_d(a, b, c_bumped)
slope_c = (d2_c - d1) / h
print(f"d(d)/d(c): {slope_c}")  # Expected: 1.0 (coefficient
   of c)
```

Listing 2: Numerical Partial Derivative Calculation

This numerical verification matches the analytical derivatives:

- $\frac{\partial d}{\partial a} = b = -3$
- $\frac{\partial d}{\partial b} = a = 2$
- $\frac{\partial d}{\partial c} = 1 = 1$

This intuition is critical: the derivative tells us the direct influence of a small change in an input on the output.

1.3 Building the Value Object in Micrograd

Micrograd's core data structure is the Value object, which wraps a scalar number and tracks how it was computed, forming an expression graph.

1.3.1 Basic Value Class Structure

The Value class holds the actual numerical data and a grad attribute, which will store the derivative of the final output (loss) with respect to this value. Initially, grad is set to zero, implying no influence on the output until gradients are computed.

```
class Value:
    def __init__(self, data, _children=(), _op=''):
        self.data = data
        self.grad = 0.0  # Initialize gradient to zero
        # Internal variables to build the expression graph
        self._prev = set(_children)  # Set of Value objects
            that are children
        self._op = _op  # String representing the operation
            that produced this value

    def __repr__(self):
        # Provides a nice string representation for printing
        return f"Value(data={self.data}, grad={self.grad})"

# Example usage
a = Value(2.0)
print(a)  # Output: Value(data=2.0, grad=0.0)
```

Listing 3: Initial Value Class

1.3.2 Overloading Operators for Graph Construction

To build mathematical expressions naturally (e.g., a + b, a * b), we overload Python's special methods (dunder methods) like __add__ and __mul__. Crucially, these methods not only perform the operation but also store the "lineage" of the new Value object: its _prev (children nodes) and _op (operation name).

```
import math

class Value:
    def __init__(self, data, _children=(), _op=''):
        self.data = data
        self.grad = 0.0
        self._prev = set(_children)
        self._op = _op
        # Stores the local backward function for this node
        self._backward = lambda: None  # Default empty
            backward function for leaf nodes

    def __repr__(self):
        return f"Value(data={self.data}, grad={self.grad})"

    def __add__(self, other):
        # Handle scalar addition (e.g., Value + 1)
        other = other if isinstance(other, Value) else Value(
            other)
        out = Value(self.data + other.data, (self, other), '+'
            )
```

```
19
20          def _backward():
21              # For addition, the gradient is simply passed
                    through (local derivative is 1)
22              self.grad += out.grad * 1.0  # Accumulate gradient
23              other.grad += out.grad * 1.0  # Accumulate
                    gradient
24          out._backward = _backward
25          return out
26
27      def __mul__(self, other):
28          # Handle scalar multiplication (e.g., Value * 2)
29          other = other if isinstance(other, Value) else Value(
                other)
30          out = Value(self.data * other.data, (self, other), '*'
                )
31
32          def _backward():
33              # For multiplication, local derivatives are the '
                    other' operand
34              self.grad += out.grad * other.data  # Accumulate
                    gradient
35              other.grad += out.grad * self.data  # Accumulate
                    gradient
36          out._backward = _backward
37          return out
38
39      # For operations like 2 * a (reversed multiplication)
40      def __rmul__(self, other):
41          return self * other
42
43      def __pow__(self, other):
44          # This implementation expects 'other' to be a scalar (
                int or float), not a Value object
45          assert isinstance(other, (int, float)), "only
                supporting int/float powers for now"
46          out = Value(self.data**other, (self,), f'**{other}')
47
48          def _backward():
49              # Power rule: d/dx(x^n) = n*x^(n-1)
50              self.grad += out.grad * (other * (self.data**(
                    other-1)))
51          out._backward = _backward
52          return out
53
54      def __neg__(self):  # -self
55          return self * -1
56
57      def __sub__(self, other):  # self - other
58          return self + (-other)
59
60      def __truediv__(self, other):  # self / other
61          # Division is implemented as multiplication by a
                negative power
62          return self * other**-1
63
```

```
64      def exp(self):
65          x = self.data
66          out = Value(math.exp(x), (self,), 'exp')
67
68          def _backward():
69              # d/dx(e^x) = e^x
70              self.grad += out.grad * out.data
71          out._backward = _backward
72          return out
73
74      def tanh(self):
75          x = self.data
76          t = (math.exp(2*x) - 1) / (math.exp(2*x) + 1)
77          out = Value(t, (self,), 'tanh')
78
79          def _backward():
80              # d/dx(tanh(x)) = 1 - tanh(x)^2
81              self.grad += out.grad * (1 - t**2)
82          out._backward = _backward
83          return out
84
85      def backward(self):
86          # Zero out all gradients first (crucial for iterative
              training)
87          # This is a common bug: forgetting to zero gradients
88          # In a real training loop, this would be handled
              globally for all parameters
89          # For this Value object, we only zero its own and its
              children's gradients for demonstration.
90          # In the full training loop, all network parameters
              would be zeroed.
91
92          topo = []
93          visited = set()
94          def build_topo(v):
95              if v not in visited:
96                  visited.add(v)
97                  for child in v._prev:
98                      build_topo(child)
99                  topo.append(v)
100         build_topo(self)
101
102         self.grad = 1.0  # Initialize the gradient of the root
              node to 1.0
103
104         for node in reversed(topo):
105             node._backward()  # Call the stored backward
                  function for each node
```

Listing 4: Adding Arithmetic Operations to Value

Important Considerations in Value Implementation

- **Accumulation of Gradients (+=)**: When a Value object is used multiple times in an expression (e.g., b = a + a), its gradient should be accumulated rather than

overwritten. This is why self.grad += ... is used instead of self.grad = ... This is a common bug if not handled correctly.

- **Handling Scalar Operands**: To allow operations like a + 1 where 1 is a Python int (not a Value object), the other operand is wrapped in a Value object if it's not already one.
- **Reversed Operations (__rmul__)**: Python's operator precedence means 2 * a would try to call 2.__mul__(a), which fails for an int. Implementing __rmul__ (reversed multiply) in Value allows Python to fall back to a.__rmul__(2), which then correctly calls a.__mul__(2).
- **Arbitrary Function _backward**: The _backward function stored in each Value object captures the local derivative calculation. This allows Micrograd to support any operation (like tanh or exp) as long as its local derivative is known and implemented.

1.3.3 Visualizing the Expression Graph

A helper function, drawdot, (not provided in source for brevity, but demonstrated) uses Graphviz to visualize the computational graph built by Micrograd, showing nodes (Value objects) and edges (operations). This helps to intuitively understand the flow of computation.

1.4 Backpropagation: The Automatic Gradient Algorithm

Backpropagation is the algorithm that efficiently calculates the gradients by recursively applying the chain rule backwards through the computation graph.

1.4.1 The Chain Rule

The chain rule is the mathematical foundation of backpropagation. It allows us to compute the derivative of a composite function. If a variable Z depends on Y, and Y depends on X, then Z also depends on X through Y. The chain rule states:

$$\frac{dZ}{dX} = \frac{dZ}{dY} \times \frac{dY}{dX}$$

Intuitively, if a car travels twice as fast as a bicycle, and the bicycle is four times as fast as a walking man, then the car travels $2 \times 4 = 8$ times as fast as the man. The "rates of change" multiply.

1.4.2 How Backpropagation Works in Practice

1. **Forward Pass**: The mathematical expression is evaluated from inputs to output, computing the data value for each Value node.
2. **Initialization**: The grad of the final output (often the loss function) is set to 1.0 (since $\frac{dL}{dL} = 1$). All other grad values are initialized to zero.
3. **Topological Sort**: The nodes of the expression graph are ordered such that a node's children always appear before it in the list (if traversed forward). For backpropagation, this list is then reversed, ensuring that when _backward is called on a node, all its dependencies (nodes "further down" the graph) have already had their grad contributions propagated.

```python
# Assuming 'self' is the root node (e.g., the loss)
topo = []
visited = set()
def build_topo(v):
    if v not in visited:
        visited.add(v)
        for child in v._prev:
            build_topo(child)
        topo.append(v)
build_topo(self)  # Populates 'topo' list

# Gradients are then computed by iterating in reverse order
for node in reversed(topo):
    node._backward()
```

Listing 5: Topological Sort Logic (within backward method)

4. **Backward Pass (_backward calls)**: Starting from the output node (gradient of 1.0) and iterating backward through the topologically sorted list, each node's _backward function is called. This function applies the chain rule: it takes the current node's grad (which is $\frac{dL}{\text{current\_node}}$) and multiplies it by the *local derivative* of how its children influenced it. The result is then *added* (+=) to the children's grad attributes.
    - For a + operation (e.g., c = a + b): $\frac{\partial c}{\partial a} = 1$, $\frac{\partial c}{\partial b} = 1$. So, a.grad += c.grad * 1.0, b.grad += c.grad * 1.0.
    - For a * operation (e.g., c = a * b): $\frac{\partial c}{\partial a} = b$, $\frac{\partial c}{\partial b} = a$. So, a.grad += c.grad * b.data, b.grad += c.grad * a.data.
    - For tanh (e.g., o = tanh(n)): $\frac{\partial o}{\partial n} = 1 - \tanh(n)^2 = 1 - o^2$. So, n.grad += o.grad * (1 - o.data**2).

1.5 Building a Neural Network (MLP)

Neural networks, specifically Multi-Layer Perceptrons (MLPs), are just complex mathematical expressions. We can build them using our Value objects.

1.5.1 The Neuron

A neuron is the fundamental building block. It takes multiple inputs (x), multiplies them by corresponding weights (w), sums these products, adds a bias (b), and passes the result through an activation function (e.g., tanh).

$$\text{output} = \text{activation\_fn}\left(\sum_i (x_i \times w_i) + b\right)$$

```python
import random

class Neuron:
    def __init__(self, nin):
        # nin: number of inputs to this neuron
```

```
        self.w = [Value(random.uniform(-1,1)) for _ in range(
            nin)]
        self.b = Value(random.uniform(-1,1))

    def __call__(self, x):
        # x: list of input Values
        # w * x + b (dot product)
        act = sum((wi*xi for wi, xi in zip(self.w, x)), self.b
            )
        out = act.tanh()  # Activation function
        return out

    def parameters(self):
        # Returns all learnable parameters (weights and bias)
        return self.w + [self.b]
```

Listing 6: Neuron Class Implementation

1.5.2 The Layer

A layer in an MLP is simply a collection of independent neurons, all receiving the same inputs from the previous layer or the network's initial input.

```
class Layer:
    def __init__(self, nin, nout):
        # nin: number of inputs to the layer (from previous
            layer or network input)
        # nout: number of neurons in this layer (number of
            outputs from this layer)
        self.neurons = [Neuron(nin) for _ in range(nout)]

    def __call__(self, x):
        # x: list of input Values from the previous layer/
            input
        # Evaluates each neuron independently
        outs = [n(x) for n in self.neurons]
        # Return a list of outputs, or single output if only
            one neuron
        return outs if len(outs) == 1 else outs

    def parameters(self):
        # Collects all parameters from all neurons in this
            layer
        return [p for neuron in self.neurons for p in neuron.
            parameters()]
```

Listing 7: Layer Class Implementation

1.5.3 The Multi-Layer Perceptron (MLP)

An MLP is a sequence of layers, where the outputs of one layer serve as the inputs to the next.

```
class MLP:
```

```
    def __init__(self, nin, nouts):
        # nin: number of inputs to the MLP
        # nouts: list of integers defining the sizes of each layer
        # Example: nouts=[4, 4, 1] creates two hidden layers of 4 neurons, and an output layer of 1 neuron
        sz = [nin] + nouts
        self.layers = [Layer(sz[i], sz[i+1]) for i in range(len(nouts))]

    def __call__(self, x):
        # x: list of input Values to the MLP
        # Feeds outputs of one layer as inputs to the next
        for layer in self.layers:
            x = layer(x)
        return x

    def parameters(self):
        # Collects all parameters from all layers in the MLP
        return [p for layer in self.layers for p in layer.parameters()]
```

Listing 8: MLP Class Implementation

1.6 Neural Network Training Loop (Gradient Descent)

The training process involves repeatedly calculating the network's output, measuring its error, and updating its weights using the gradients obtained via backpropagation.

1.6.1 The Loss Function

The loss function quantifies how "bad" the neural network's predictions are compared to the desired targets (ground truth). The goal of training is to minimize this loss. A common example is Mean Squared Error (MSE) loss:

$$L = \frac{1}{N}\sum_{i=1}^{N}(y_{\text{pred},i} - y_{\text{true},i})^2$$

where y_{pred} are the network's predictions and y_{true} are the actual targets.

```
# Example dataset (4 inputs, 4 targets for binary classification)
xs = [
    [2.0, 3.0, -1.0],
    [3.0, -1.0, 0.5],
    [0.5, 1.0, 1.0],
    [1.0, 1.0, -1.0],
]
ys = [1.0, -1.0, -1.0, 1.0]  # Desired targets

# Initialize a sample MLP
# 3 inputs -> 2 hidden layers of 4 neurons each -> 1 output neuron
n = MLP(3, [4, 4, 1])
```

```
13
14 # Forward pass to get predictions
15 ypred = [n(x) for x in xs]
16 print("Initial predictions:", [p.data for p in ypred])
17
18 # Calculate Mean Squared Error loss
19 loss = sum([(yout - ygt)**2 for ygt, yout in zip(ys, ypred)])
20 print("Initial Loss:", loss.data)
```

Listing 9: Example Data and Loss Calculation

1.6.2 The Training Loop Steps

The core training loop for gradient descent consists of these iterative steps:

1. **Forward Pass**: Feed the input data through the neural network to obtain predictions (ypred).
2. **Calculate Loss**: Compare ypred with ys (targets) using the chosen loss function to get a single loss value.
3. **Zero Gradients**: Before performing backpropagation for the current step, it is crucial to reset all accumulated gradients in the network's parameters to zero. Forgetting this is a common bug, as gradients from previous steps would otherwise incorrectly accumulate.
4. **Backward Pass**: Call loss.backward(). This propagates the gradient of the loss all the way back through the network, filling the grad attribute of every Value object, especially the parameters (weights w and biases b).
5. **Update Parameters**: Adjust each parameter's data value by taking a small step in the direction opposite to its gradient. This is done to minimize the loss.

$$p.\text{data} \leftarrow p.\text{data} - \text{learning\_rate} \times p.\text{grad}$$

Here, learning_rate (or step_size) is a small positive scalar that controls the magnitude of the update.

```
1  # Re-initialize the network for a fresh start
2  n = MLP(3, [4, 4, 1])
3
4  # Training hyperparameters
5  num_epochs = 50  # Number of training iterations
6  learning_rate = 0.05
7
8  for k in range(num_epochs):
9      # 1. Forward pass
10     ypred = [n(x) for x in xs]
11
12     # 2. Calculate loss
13     loss = sum([(yout - ygt)**2 for ygt, yout in zip(ys, ypred
           )])
14
15     # 3. Zero gradients (VERY IMPORTANT!)
16     for p in n.parameters():
```

```
        p.grad = 0.0

    # 4. Backward pass
    loss.backward()

    # 5. Update parameters (gradient descent step)
    for p in n.parameters():
        p.data += -learning_rate * p.grad

    # Print progress
    print(f"Epoch {k}: Loss = {loss.data:.4f}")

# Final predictions after training
print("\nFinal predictions:", [p.data for p in ypred])
print("Desired targets:", ys)
```

Listing 10: Full Training Loop Example

1.6.3 Learning Rate and Stability

The learning_rate (or step_size) is a critical hyperparameter.

- If too low, training will be very slow and may take too long to converge.
- If too high, the optimization can become unstable, oscillate, or even "explode" the loss because it oversteps the optimal direction indicated by the local gradient.

Finding the right learning rate is often an art, though more advanced optimizers automate some of this tuning.

1.7 Micrograd vs. PyTorch: A Comparison

Micrograd's design directly mirrors the core functionalities of modern deep learning frameworks like PyTorch, allowing for a deep intuitive understanding before dealing with production complexities.

1.7.1 Similarities in API and Core Concepts

- **Value vs. torch.Tensor**: Both wrap numerical data and contain a .grad attribute to store gradients. PyTorch's Tensor is an N-dimensional array of scalars, while Micrograd's Value is strictly scalar-valued.
- **Graph Construction**: Both frameworks build a computation graph implicitly as operations are performed on their respective data structures (Value or Tensor).
- **.backward() Method**: Both provide a .backward() method on the root node (e.g., loss) to trigger the backpropagation algorithm.
- **_backward / local_derivative**: The principle of defining how to backpropagate through each atomic operation (by providing its local derivative) is fundamental to both. In PyTorch, custom functions require implementing both a forward and backward method.
- **zero_grad()**: The necessity to reset gradients before a new backward pass is a core concept in both Micrograd (manually implemented) and PyTorch (e.g., via optimizer.zero_grad() or model.zero_grad()).

```python
import torch

# Similar inputs and weights as in Micrograd neuron example
x1 = torch.tensor(2.0, requires_grad=True)
w1 = torch.tensor(-3.0, requires_grad=True)
x2 = torch.tensor(0.0, requires_grad=True)
w2 = torch.tensor(1.0, requires_grad=True)
b = torch.tensor(6.8813735870195432, requires_grad=True)

# Graph construction (forward pass)
n = x1*w1 + x2*w2 + b
o = n.tanh()

print(f"PyTorch Forward Pass: {o.item():.7f}")

# Backward pass
o.backward()

# Gradients
print(f"PyTorch Gradients:")
print(f" x1.grad: {x1.grad.item():.7f}")  # Corresponds to x1.grad in Micrograd
print(f" w1.grad: {w1.grad.item():.7f}")  # Corresponds to w1.grad in Micrograd
print(f" x2.grad: {x2.grad.item():.7f}")  # Corresponds to x2.grad in Micrograd
print(f" w2.grad: {w2.grad.item():.7f}")  # Corresponds to w2.grad in Micrograd
print(f" b.grad: {b.grad.item():.7f}")   # Corresponds to b.grad in Micrograd

# PyTorch output will match Micrograd's (0.7071067, -1.5000000, 1.0000000, 0.0000000, 0.5000000, 0.5000000)
```

Listing 11: PyTorch Equivalent Example for a Neuron

1.7.2 Key Differences and Production-Grade Features

- **Tensors and Efficiency**: PyTorch's primary advantage is its use of tensors, which enable highly optimized, parallelized operations on GPUs, making computations vastly faster than scalar operations for large datasets.
- **Graph Complexity**: Production libraries manage extremely large and dynamic computation graphs, whereas Micrograd's graph construction and traversal are simpler due to its scalar nature.
- **Advanced Optimizers**: While Micrograd uses basic gradient descent, PyTorch offers a wide array of sophisticated optimizers (e.g., Adam, RMSprop) that adapt the learning rate during training.
- **Loss Functions**: PyTorch includes many specialized loss functions (e.g., Cross-Entropy Loss for classification, Max Margin Loss), often with built-in numerical stability features.
- **Batching and Regularization**: PyTorch supports processing data in batches

(subsets of the full dataset) for efficiency with large datasets and includes features like L2 regularization to prevent overfitting.

- **Learning Rate Schedules**: Advanced techniques like learning rate decay, where the learning rate decreases over time, are common in PyTorch to stabilize training towards the end.
- **Implementation Details**: Finding the exact backward pass code for specific operations in a production library like PyTorch can be challenging due to the sheer size and complexity of the codebase, which is highly optimized for performance across different hardware (CPU/GPU kernels) and data types.

1.8 Conclusion

This lecture has provided a comprehensive understanding of neural network training "under the hood" through the lens of Micrograd. We've seen that:

- Neural networks are complex mathematical expressions.
- Training involves minimizing a loss function through iterative gradient descent.
- Backpropagation, a recursive application of the chain rule, efficiently computes these gradients.
- The core Value object and its overloaded operators build the computational graph.
- Understanding the intuitive meaning of derivatives and the chain rule is paramount.

Micrograd, despite its simplicity, demonstrates the fundamental principles that power even the largest neural networks with billions or trillions of parameters used in complex applications like GPT models. The core concepts of forward pass, backward pass (gradient calculation), and parameter updates remain identical, regardless of scale.