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
```

```
    def exp(self):
        x = self.data
        out = Value(math.exp(x), (self,), 'exp')

        def _backward():
            # d/dx(e^x) = e^x
            self.grad += out.grad * out.data
        out._backward = _backward
        return out

    def tanh(self):
        x = self.data
        t = (math.exp(2*x) - 1) / (math.exp(2*x) + 1)
        out = Value(t, (self,), 'tanh')

        def _backward():
            # d/dx(tanh(x)) = 1 - tanh(x)^2
            self.grad += out.grad * (1 - t**2)
        out._backward = _backward
        return out

    def backward(self):
        # Zero out all gradients first (crucial for iterative
            training)
        # This is a common bug: forgetting to zero gradients
        # In a real training loop, this would be handled
            globally for all parameters
        # For this Value object, we only zero its own and its
            children's gradients for demonstration.
        # In the full training loop, all network parameters
            would be zeroed.

        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)

        self.grad = 1.0  # Initialize the gradient of the root
            node to 1.0

        for node in reversed(topo):
            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.

```
# 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).

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

```
import random

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

```
6           self.w = [Value(random.uniform(-1,1)) for _ in range(
                nin)]
7           self.b = Value(random.uniform(-1,1))
8
9       def __call__(self, x):
10          # x: list of input Values
11          # w * x + b (dot product)
12          act = sum((wi*xi for wi, xi in zip(self.w, x)), self.b
                )
13          out = act.tanh()  # Activation function
14          return out
15
16      def parameters(self):
17          # Returns all learnable parameters (weights and bias)
18          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.

```
1   class Layer:
2       def __init__(self, nin, nout):
3           # nin: number of inputs to the layer (from previous
                layer or network input)
4           # nout: number of neurons in this layer (number of
                outputs from this layer)
5           self.neurons = [Neuron(nin) for _ in range(nout)]
6
7       def __call__(self, x):
8           # x: list of input Values from the previous layer/
                input
9           # Evaluates each neuron independently
10          outs = [n(x) for n in self.neurons]
11          # Return a list of outputs, or single output if only
                one neuron
12          return outs if len(outs) == 1 else outs
13
14      def parameters(self):
15          # Collects all parameters from all neurons in this
                layer
16          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.

```
1   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.

2 Lecture 2: The Spelled-Out Intro to Language Modeling (Makemore)

Abstract

Welcome to these comprehensive lecture notes on building 'makemore', a project designed to illustrate the fundamentals of language modeling. Just as 'micrograd' was built step-by-step to demystify automatic differentiation, 'makemore' will be constructed slowly and thoroughly to explain character-level language models, all the way up to the architecture of modern transformers like GPT-2, at the character level.

2.1 Introduction to Makemore

'Makemore' is a system that, as its name suggests, "makes more" of the type of data it is trained on. For instance, if provided with a dataset of names, it learns to generate new sequences that sound like names but are unique. The primary dataset used for demonstration is 'names.txt', which contains approximately 32,000 names collected from a government website. After training, 'makemore' can generate unique names such as "Dontel," "Irot," or "Zhendi," which sound plausible but are not real names from the dataset.

2.1.1 Character-Level Language Models

At its core, ‘makemore‘ operates as a **character-level language model**. This means it processes each line in the dataset (e.g., a name) as a sequence of individual characters. The model’s primary task is to learn the statistical relationships between characters to predict the next character in a sequence. This foundational understanding will then be extended to word-level models for document generation, and eventually to image and image-text networks like DALL-E and Stable Diffusion.

2.2 Building a Bigram Language Model: The Statistical Approach

We begin our journey by implementing a very simple character-level language model: the **bigram language model**. In a bigram model, the prediction of the next character is based solely on the immediately preceding character, ignoring any information further back in the sequence. This is a simple yet effective starting point to grasp the core concepts.

2.2.1 Data Loading and Preparation

The first step is to load the ‘names.txt‘ dataset.

```
import torch # We'll use PyTorch later, but good to import early.

# Load the dataset
words = open('names.txt', 'r').read().splitlines()

# Display basic statistics
print(f"Total words: {len(words)}") # Expected: ~32,000
print(f"Shortest word: {min(len(w) for w in words)}") # Expected: 2
print(f"Longest word: {max(len(w) for w in words)}") # Expected: 15
print(f"First 10 words: {words[:10]}")
```

Listing 12: Loading and preparing the dataset

2.2.2 Identifying Bigrams and Special Tokens

Each word, like ”isabella,” implicitly contains several bigram examples. For instance:

- ’i’ is likely to come first.
- ’s’ is likely to follow ’i’.
- ’a’ is likely to follow ’is’.
- ...and so on.
- A special piece of information is that after ”isabella,” the word is likely to end.

To capture these start and end conditions, we introduce a special token, ‘.‘ (dot), to represent both the start and end of a word. This simplified approach uses a single special token instead of separate start/end tokens, which is more pleasing and efficient.

For a word like ”emma”, the bigrams would be: ‘(.e)‘, ‘(e,m)‘, ‘(m,m)‘, ‘(m,a)‘, ‘(a,.)‘.

```
# Example for a single word
word = "emma"
chars = ['.'] + list(word) + ['.'] # Add start/end tokens
for ch1, ch2 in zip(chars, chars[1:]):
    print(ch1, ch2)
```

Listing 13: Extracting Bigrams from a word with special tokens

2.2.3 Counting Bigram Frequencies

The simplest way to learn the statistics of which characters follow others is by counting their occurrences in the dataset. Initially, we can use a Python dictionary to store these counts, mapping each bigram (as a tuple of two characters) to its frequency.

```
b = {} # Our dictionary for counts
for w in words:
    chars = ['.'] + list(w) + ['.']
    for ch1, ch2 in zip(chars, chars[1:]):
        bigram = (ch1, ch2)
        b[bigram] = b.get(bigram, 0) + 1 # Increment count,
            default to 0 if new

# Sort and display most common bigrams
sorted_b = sorted(b.items(), key=lambda kv: kv[1], reverse=
    True)
print(f"Top 10 most common bigrams: {sorted_b[:10]}")
# Example: (('n', '.'), 7000) means 'n' is followed by end-
    token 7000 times
# Example: (('a', 'n'), 6000) means 'a' is followed by 'n'
    6000 times
```

Listing 14: Counting Bigram Frequencies with a Dictionary

2.2.4 Transition to a PyTorch Tensor for Counts

While a dictionary works, it is significantly more convenient and efficient to store these counts in a two-dimensional array, specifically a PyTorch tensor. The rows will represent the first character of a bigram, and the columns will represent the second character. Each entry ‘N[row, col]‘ will indicate how often ‘col‘ follows ‘row‘.

First, we need a mapping from characters to integers (s2i) and vice-versa (i2s). We will place the special ‘.‘ token at index 0, and subsequent letters (a-z) will be mapped to indices 1-26.

```
# Create a list of all unique characters, sorted, and include
    the special '.' token
chars = sorted(list(set(''.join(words))))
s2i = {s:i+1 for i,s in enumerate(chars)} # Map a-z to 1-26
s2i['.'] = 0 # Map '.' to 0
i2s = {i:s for s,i in s2i.items()} # Reverse mapping

print(f"Character to index mapping (s2i): {s2i}")
print(f"Index to character mapping (i2s): {i2s}")
```

Listing 15: Character-to-Integer Mapping

Now, we can populate our 2D PyTorch tensor:

```
# Create a 27x27 tensor of zeros (26 letters + 1 special char)
N = torch.zeros((27, 27), dtype=torch.int32) # Use int32 for
    counts

for w in words:
    chars = ['.'] + list(w) + ['.']
    for ch1, ch2 in zip(chars, chars[1:]):
        ix1 = s2i[ch1]
        ix2 = s2i[ch2]
        N[ix1, ix2] += 1 # Increment count in the tensor

# Visualize a small part of the N matrix (e.g., first few rows
    /cols)
print("Shape of N:", N.shape)
print("N[0, :] (counts for characters following '.'):", N[0,
    :]) # First row shows start probabilities
```

Listing 16: Populating the Count Matrix N

The ‘N‘ matrix visually represents the statistical structure, showing how often certain characters follow others. The row at index 0 (for ‘.‘) indicates the counts for the first letters of names, and the column at index 0 (for ‘.‘) indicates counts for letters preceding the end of a name.

2.2.5 Converting Counts to Probabilities

To use the bigram model for prediction, we need to convert the raw counts in ‘N‘ into probabilities. This is done by normalizing each row of the ‘N‘ matrix such that the sum of probabilities in each row equals 1.

```
# Convert N to float and normalize each row
P = (N+1).float() # Add 1 for smoothing (explained later) and
    convert to float
P /= P.sum(1, keepdim=True) # Divide each row by its sum to
    get probabilities

# Check normalization for the first row (should sum to 1)
print(f"Sum of probabilities in first row P: {P[0].sum()}")
```

Listing 17: Converting Counts to Probabilities (P)

Broadcasting Semantics Note: When performing operations like ‘P /= P.sum(1, keepdim=True)‘, PyTorch applies ”broadcasting”. ‘P‘ is 27x27, and ‘P.sum(1, keepdim=True)‘ results in a 27x1 column vector. PyTorch automatically stretches this 27x1 vector across the columns of ‘P‘ (replicating it 27 times) to enable element-wise division, effectively normalizing each row independently. It is crucial to use ‘keepdim=True‘ to maintain the dimension for correct broadcasting and avoid subtle bugs where operations might occur in an unintended direction.

2.2.6 Sampling from the Bigram Model

With the probability matrix 'P', we can now generate new sequences. The process is iterative: 1. Start with the special '.' token (index 0). 2. Look at the row in 'P' corresponding to the current character. 3. Sample the next character based on the probabilities in that row using 'torch.multinomial'. 4. Append the sampled character to the generated sequence. 5. If the sampled character is the '.' token, the word ends; otherwise, repeat from step 2.

```
g = torch.Generator().manual_seed(2147483647) # For
    reproducibility

for _ in range(10): # Generate 10 names
    out = []
    ix = 0 # Start with '.' token

    while True:
        p = P[ix] # Get the probability distribution for the
            current character
        ix = torch.multinomial(p, num_samples=1, replacement=
            True, generator=g).item() # Sample next char
        if ix == 0: # If we sampled '.', it's the end of the
            word
            break
        out.append(i2s[ix]) # Add character to the output list
    print(''.join(out)) # Join characters to form the name
```

Listing 18: Sampling from the Bigram Model

The generated names may seem "terrible" (e.g., "h", "yanu", "o'reilly"). This is because the bigram model is very simple; it only considers the immediate preceding character and has no long-term memory or understanding of name structure beyond two-character sequences.

2.3 Evaluating Model Quality: The Loss Function

To quantify how "good" our model is, we need a single number that summarizes its quality. This is typically done using a **loss function**, which we aim to minimize.

2.3.1 Likelihood and Log-Likelihood

For a language model, the quality is often measured by its **likelihood** of generating the observed training data. This is calculated as the product of the probabilities that the model assigns to each bigram in the training set. A higher likelihood indicates a better model.

However, multiplying many probabilities (which are between 0 and 1) results in extremely small, unwieldy numbers. To overcome this, we use the **log-likelihood**, which is the sum of the logarithms of the individual probabilities.

Mathematically, if $L = p_1 \times p_2 \times \cdots \times p_n$ (likelihood), then $\log(L) = \log(p_1) + \log(p_2) + \cdots + \log(p_n)$ (log-likelihood).

The logarithm is a monotonic transformation, meaning maximizing the likelihood is equivalent to maximizing the log-likelihood. Logarithms are also helpful because probabilities near 1 yield log probabilities near 0, while probabilities near 0 yield large negative

log probabilities.

2.3.2 Negative Log-Likelihood (NLL) as Loss

For optimization, we prefer a loss function where "lower is better". Therefore, we transform the log-likelihood into the **negative log-likelihood (NLL)** by simply taking its negative value.

The lowest possible NLL is 0 (when all probabilities assigned by the model to the correct next characters are 1), and it grows positive as the model's predictions worsen.

For convenience and comparison, we often normalize this sum by the total number of bigrams, resulting in the **average negative log-likelihood**.

```
log_likelihood = 0.0
n = 0 # Count of bigrams

for w in words:
    chars = ['.'] + list(w) + ['.']
    for ch1, ch2 in zip(chars, chars[1:]):
        ix1 = s2i[ch1]
        ix2 = s2i[ch2]
        prob = P[ix1, ix2] # Probability model assigns to this
            bigram
        logprob = torch.log(prob) # Log of that probability
        log_likelihood += logprob # Sum log probabilities
        n += 1 # Count bigram

# Average Negative Log-Likelihood
nll = -log_likelihood
average_nll = nll / n
print(f"Total bigrams: {n}")
print(f"Negative Log Likelihood: {nll:.4f}")
print(f"Average Negative Log Likelihood (Loss): {average_nll
    :.4f}") # Expected: ~2.45 after smoothing
```

Listing 19: Calculating Average Negative Log-Likelihood Loss

The goal of training is to minimize this average NLL loss.

2.3.3 Model Smoothing

A significant problem arises if the model assigns a zero probability to a bigram that actually appears in the dataset. This causes the log-probability to become negative infinity and the NLL loss to become positive infinity, making optimization impossible. This happens when a specific character sequence (e.g., 'jq' in "andrej") never occurred in the training data, so its count is 0.

To fix this, we apply **model smoothing**, specifically "add-1 smoothing" (also known as Laplace smoothing). This involves adding a small "fake count" (e.g., 1) to every bigram count before normalization. This ensures that no bigram ever has a zero count, thus preventing zero probabilities and infinite loss.

```
# Original P calculation: P = N.float() / N.sum(1, keepdim=
    True)
# With smoothing, N is incremented by 1 before normalization:
P = (N + 1).float() # Add 1 to all counts
```

```
P /= P.sum(1, keepdim=True) # Normalize as before
```

Listing 20: Model Smoothing by adding 1 to all counts

Adding more to the counts (e.g., 5, 10, or more) results in a "smoother" or more uniform probability distribution, as it biases the model towards more uniform predictions. Conversely, adding less leads to a more "peaked" distribution, closely reflecting the observed frequencies.

2.4 Building a Bigram Language Model: The Neural Network Approach

Now, we shift our perspective from explicit counting to using a neural network to learn these bigram probabilities. The goal is to arrive at a very similar model but using the powerful framework of gradient-based optimization.

2.4.1 Neural Network Architecture

Our initial neural network is the simplest possible: a single **linear layer**. It takes a single character as input and outputs a probability distribution over the 27 possible next characters.

- **Input**: A single character (represented as an integer index).
- **Neural Network (Parameters W)**: A linear transformation.
- **Output**: 27 numbers, which will be transformed into a probability distribution for the next character.

The optimization process will involve tuning the parameters (weights W) of this neural network to minimize the negative log-likelihood loss, ensuring it assigns high probabilities to the correct next characters in the training data.

2.4.2 Preparing Data for the Neural Network

The training data for the neural network consists of pairs of (input character, target character), where both are integer indices.

```
xs, ys = [], [] # Lists to store input and target indices

for w in words:
    chars = ['.'] + list(w) + ['.']
    for ch1, ch2 in zip(chars, chars[1:]):
        ix1 = s2i[ch1]
        ix2 = s2i[ch2]
        xs.append(ix1) # Input character index
        ys.append(ix2) # Target (label) character index

xs = torch.tensor(xs) # Convert to PyTorch tensor
ys = torch.tensor(ys) # Convert to PyTorch tensor

num_examples = xs.nelement() # Total number of bigram examples
print(f"Number of examples: {num_examples}")
print(f"Shape of inputs (xs): {xs.shape}, dtype: {xs.dtype}")
print(f"Shape of targets (ys): {ys.shape}, dtype: {ys.dtype}")
```

Listing 21: Creating Input (xs) and Target (ys) Tensors

2.4.3 Input Encoding: One-Hot Vectors

Neural networks don't typically take raw integer indices as input for their weights to act multiplicatively. Instead, integer inputs are commonly encoded using **one-hot encoding**. A one-hot encoded vector is a vector of zeros except for a single dimension (corresponding to the integer's value), which is set to one.

```
import torch.nn.functional as F # Common import for functional
    operations

# Encode xs into one-hot vectors. num_classes is 27 (26
    letters + '.')
# Ensure dtype is float32 for neural network operations
x_encoded = F.one_hot(xs, num_classes=27).float()
print(f"Shape of x_encoded: {x_encoded.shape}") # Expected: (
    num_examples, 27)
print(f"Dtype of x_encoded: {x_encoded.dtype}") # Should be
    torch.float32
```

Listing 22: One-Hot Encoding Inputs

2.4.4 The Forward Pass

The forward pass describes how the neural network transforms its inputs into outputs (probabilities).

1. **Initialize Weights (W)**: The single linear layer has weights W. Since there are 27 possible input characters and 27 possible output characters (probabilities for the next character), the weight matrix 'W' will be of size 27x27. It is initialized with random numbers from a normal distribution.

```
    g = torch.Generator().manual_seed(2147483647) # For
        reproducibility
    W = torch.randn((27, 27), generator=g, requires_grad=True)
        # 27x27 weights
    print(f"Shape of W: {W.shape}")
```

Listing 23: Initializing Weights

2. **Calculate Logits**: The core of the linear layer is a matrix multiplication: '$x_encoded@W$'.$Thisoperatio$

```
    # x_encoded is (num_examples, 27), W is (27, 27)
    # The result 'logits' will be (num_examples, 27)
    logits = x_encoded @ W # Matrix multiplication
    print(f"Shape of logits: {logits.shape}")
```

Listing 24: Calculating Logits

Crucially, because '$x_encoded$'$isone-hot$,'$x_encoded@W$'$effectively$"$plucksout$"$therowof$'W'$correspond$
$hotinputvector.Thismeans$'$logits[i]$'$(forthei-theexample)willbeidenticalto$'$W[ix1]$',$where$'$ix1$'$istheintege$

3. **Convert Logits to Probabilities (Softmax)**: Logits can be any real number (positive or negative). To transform them into a valid probability distribution (positive numbers that sum to 1), we use the **softmax** function. Softmax involves two steps: * **Exponentiation**: counts = e^{logits}. This converts log-counts into positive "fake counts". * **Normalization**: probabilities = $\frac{\text{counts}}{\sum \text{counts}}$. Each row of counts is normalized to sum to 1, producing probabilities.

```
    # Exponentiate logits to get "counts" (positive values)
    counts = logits.exp() # Element-wise exponentiation

    # Normalize counts to get probabilities (each row sums to
        1)
    probs = counts / counts.sum(1, keepdim=True) # Same
        broadcasting as before

    print(f"Shape of probabilities (probs): {probs.shape}") #
        (num_examples, 27)
    print(f"Sum of first row of probs: {probs[0].sum()}") #
        Should be ~1
```

Listing 25: Softmax Transformation

This entire sequence (‘logits -¿ counts -¿ probs‘) is what is commonly referred to as the **softmax layer** in neural networks. It ensures the neural network's outputs are interpretable as probability distributions. All these operations are differentiable, which is crucial for backpropagation.

2.4.5 Loss Calculation for Neural Networks

The loss function for the neural network is still the average negative log-likelihood. We need to "pluck out" the probability that the model assigned to the *correct* next character (the ‘ys‘ target) for each input example.

```
# Select the probabilities corresponding to the correct target
    characters
# torch.arange(num_examples) creates indices for each row: 0,
    1, 2, ...
# ys contains the column index (target character) for each row
correct_probs = probs[torch.arange(num_examples), ys] # Shape:
    (num_examples,)

# Calculate log probabilities and then the negative mean (
    average NLL)
loss = -correct_probs.log().mean()
print(f"Neural Network Loss (average NLL): {loss.item():.4f}")
    # .item() extracts scalar from tensor
```

Listing 26: Calculating Neural Network Loss

A high loss value (e.g., 3.76 initially) indicates that the randomly initialized network is assigning low probabilities to the correct next characters.

2.4.6 The Backward Pass and Optimization

The core idea of training a neural network is to iteratively adjust its parameters (the weights W) to minimize the loss. This is achieved using **gradient-based optimization**, specifically **gradient descent**.

1. **Zero Gradients**: Before computing new gradients, any accumulated gradients from previous iterations must be reset to zero.

```
    W.grad = None # More efficient than W.grad.zero_() in
        PyTorch
```

Listing 27: Zeroing Gradients

2. **Backpropagation**: PyTorch automatically builds a computational graph during the forward pass, tracking all operations and their dependencies. Calling 'loss.backward()' initiates backpropagation, computing the gradients of the 'loss' with respect to all tensors that 'requires$_g rad = True$'$(in our case, 'W')$. These gradients are then stored in the '.grad' attribute of 'W'.

```
    loss.backward() # Computes gradients of loss wrt W
    print(f"Shape of W.grad: {W.grad.shape}")
```

Listing 28: Backpropagation

The 'W.grad' tensor contains information on how each weight in 'W' influences the 'loss'. A positive gradient means increasing that weight would increase the loss, while a negative gradient means increasing that weight would decrease the loss.

3. **Parameter Update**: We update the weights by nudging them in the opposite direction of their gradients. This is the core of gradient descent. The 'learning$_r ate$'$(e.g., 0.1 or 50)$ controls the siz

```
    learning_rate = 50.0 # Example learning rate
    W.data += -learning_rate * W.grad # Nudge weights in
        direction of decreasing loss
```

Listing 29: Updating Weights

This process of forward pass, loss calculation, backward pass, and parameter update is repeated for many iterations (epochs).

```
g = torch.Generator().manual_seed(2147483647) # For
    reproducibility
W = torch.randn((27, 27), generator=g, requires_grad=True) #
    Initialize W

learning_rate = 50.0
num_iterations = 100 # How many steps of gradient descent

for k in range(num_iterations):
    # Forward pass:
    x_encoded = F.one_hot(xs, num_classes=27).float()
    logits = x_encoded @ W
    counts = logits.exp()
    probs = counts / counts.sum(1, keepdim=True)
    correct_probs = probs[torch.arange(num_examples), ys]
    loss = -correct_probs.log().mean()

```

```
    # Backward pass:
    W.grad = None # Zero gradients
    loss.backward() # Compute gradients

    # Update weights:
    W.data += -learning_rate * W.grad

    if k % 10 == 0:
        print(f"Iteration {k}: Loss = {loss.item():.4f}")

print(f"Final Neural Network Loss: {loss.item():.4f}") # Should be similar to ~2.45
```

Listing 30: Training Loop (Gradient Descent)

After sufficient training iterations, the neural network's loss converges to a value very similar to what was achieved with the explicit counting method (around 2.45-2.47). This is because for a bigram model, the direct counting method *is* the optimal solution for minimizing this loss function, and gradient descent finds that same optimum. The 'W' matrix, after optimization, becomes essentially the 'log(N+1)' matrix from the statistical approach, demonstrating the equivalence.

2.4.7 Model Smoothing (Regularization in Neural Nets)

In the neural network framework, the equivalent of adding "fake counts" for model smoothing is achieved through **regularization**. Specifically, adding a term to the loss function that penalizes large or non-zero weights (e.g., L2 regularization, which adds 'W.square().mean()' to the loss).

If 'W' has all its entries equal to zero, then 'logits' will be all zeros, 'counts' will be all ones, and 'probs' will be uniform (each character having equal probability). By adding a regularization loss that pushes 'W' towards zero, we incentivize smoother (more uniform) probability distributions.

```
# Add a regularization term to the loss function
# lambda_reg controls the strength of regularization
lambda_reg = 0.01 # Example regularization strength
# Original loss: -correct_probs.log().mean()
loss = -correct_probs.log().mean() + lambda_reg * (W**2).mean()
```

Listing 31: L2 Regularization for Smoothing

This regularization term acts like a "spring force" pulling the weights towards zero, balancing the data-driven loss that tries to match the observed probabilities. A stronger regularization 'lambda$_reg$' $leads to a smoother model, analogous to adding more fake counts.$

2.4.8 Sampling from the Neural Network Model

Once the neural network is trained, sampling new names works exactly as with the statistical bigram model. The difference is that the probability distribution 'p' for the next character is now computed by passing the current character through the trained neural network (forward pass), rather than looking it up in the pre-computed 'P' table.

```
g = torch.Generator().manual_seed(2147483647 + 10) # Different seed for different samples

for _ in range(10): # Generate 10 names
    out = []
    ix = 0 # Start with '.' token
    while True:
        # Forward pass through the neural net to get probabilities
        x_encoded = F.one_hot(torch.tensor([ix]), num_classes=27).float() # Input single character
        logits = x_encoded @ W # Logits for current char
        counts = logits.exp() # Counts
        p = counts / counts.sum(1, keepdim=True) # Probabilities

        ix = torch.multinomial(p, num_samples=1, replacement=True, generator=g).item() # Sample
        if ix == 0:
            break
        out.append(i2s[ix])
    print(''.join(out))
```

Listing 32: Sampling from the Trained Neural Network

Since the trained neural network effectively learned the same underlying probability distribution as the counting method, it produces identical-looking samples and achieves the same loss.

2.5 Conclusion and Future Extensions

We have built and explored a bigram character-level language model using two distinct approaches:

1. **Statistical Counting**: Directly counting bigram frequencies and normalizing them to form a probability distribution matrix.
2. **Neural Network (Gradient-Based Optimization)**: Using a simple linear layer, one-hot encoding, and softmax to produce probabilities, then optimizing weights with gradient descent to minimize negative log-likelihood loss.

Both methods lead to the same model and results for the bigram case.

The true power of the neural network approach lies in its scalability and flexibility. While the counting method is simple for bigrams, it becomes intractable for longer sequences (e.g., if we consider the last 10 characters to predict the next), as the number of possible combinations explodes, making a lookup table infeasible.

In future developments, this framework will be expanded:

- Taking more previous characters as input (not just one).
- Using increasingly complex neural network architectures, moving beyond a single linear layer to multi-layer perceptrons, recurrent neural networks, and ultimately, modern **transformers** (like GPT-2's core mechanism).

Despite this increasing complexity, the fundamental principles of the forward pass (producing logits, softmax to probabilities), loss calculation (negative log-likelihood), and optimization (gradient descent) will remain consistent.

3 Lecture 3: Building Makemore Part 2 - Multi-Layer Perceptron (MLP)

Abstract

Welcome to the second installment of our "makemore" series! In this lecture, we transition from simpler models to a more sophisticated neural network approach to improve our character-level language modeling. Our goal is to generate more name-like sequences by considering greater context when predicting the next character.

3.1 Limitations of the Bigram Model and the Need for MLPs

In the previous lecture, we implemented a bigram language model, which predicted the next character based solely on the immediately preceding character. This was done using both counts and a simple neural network with a single linear layer.

While approachable, the bigram model suffered from a significant limitation: it only considered one character of context. This severely limited its predictive power, leading to generated names that didn't sound very realistic.

The core problem with extending this count-based approach to more context (e.g., trigrams or longer) is that the size of the required lookup table (or matrix of counts) grows exponentially with the context length.

- 1 character context: 27 possibilities.
- 2 characters context: $27 \times 27 = 729$ possibilities.
- 3 characters context: $27 \times 27 \times 27 \approx 20,000$ possibilities.

This exponential growth quickly leads to an impractically large matrix with too few counts for each possibility, causing the model to "blow up" and perform poorly.

To overcome this, we adopt a Multi-Layer Perceptron (MLP) model, inspired by the influential paper by Bengio et al. (2003).

3.2 The Bengio et al. (2003) Modeling Approach

The Bengio et al. (2003) paper was highly influential in demonstrating the use of neural networks for predicting the next token in a sequence, specifically focusing on a word-level language model with a vocabulary of 17,000 words. While their paper focuses on words, we apply the same core modeling approach to characters.

3.2.1 Core Idea: Word/Character Embeddings

The central innovation is associating a low-dimensional "feature vector" (an embedding) to each word or character in the vocabulary.

- For 17,000 words, they embedded each into a 30-dimensional space, creating 17,000 vectors in this space.
- Initially, these embeddings are randomly initialized and spread out.
- During neural network training, these embedding vectors are tuned using backpropagation, causing them to move around in the space.
- The intuition is that words with similar meanings (or synonyms) will end up in similar parts of the embedding space, while unrelated words will be far apart.

3.2.2 Generalization through Embeddings

This embedding approach facilitates generalization to novel scenarios.

- **Example:** If the phrase "a dog was running in a [blank]" has never been seen, but "the dog was running in a [blank]" has, the network can still make a good prediction.
- This is because the embeddings for "a" and "the" might be learned to be close to each other, allowing knowledge to transfer.
- Similarly, if "cats" and "dogs" co-occur in similar contexts, their embeddings will be close, enabling the model to generalize even if it hasn't seen the exact phrase with one or the other.

3.2.3 Neural Network Architecture

The core modeling approach involves a multi-layer neural network to predict the next word/character given previous ones, trained by maximizing the log likelihood of the training data.

The network diagram for predicting the fourth word given three previous words is as follows:

1. **Input Layer (Embedding Lookup Table C):**
    - Each of the three previous words (or characters in our case) is represented by an integer index from the vocabulary (e.g., 0 to 16999 for 17,000 words).
    - These indices are fed into a shared "lookup table" (matrix C).
    - Matrix C has dimensions 'Vocabulary Size x Embedding Dimension' (e.g., 17,000 x 30).
    - Each integer index "plucks out" a corresponding row from C, converting the index into its dense embedding vector (e.g., a 30-dimensional vector for each word).
    - If we have three previous words, and each word has a 30-dimensional embedding, the combined input to the next layer is 90 neurons (3×30).
2. **Hidden Layer:**
    - This is a fully connected layer.
    - The size of this layer (number of neurons) is a 'hyperparameter' (a design choice, e.g., 100 neurons).
    - It takes the concatenated embeddings from the input layer (e.g., 90 numbers) and transforms them.
    - A 'tanh' non-linearity is applied to the output of this layer.
3. **Output Layer:**
    - This is also a fully connected layer.
    - It has 'Vocabulary Size' neurons (e.g., 17,000 for words, or 27 for characters).
    - This layer is typically the most computationally expensive due to the large number of parameters when dealing with large vocabularies.
4. **Softmax Layer:**
    - The outputs of the final layer ("logits") are passed through a 'softmax' function.

- Softmax exponentiates each logit and normalizes them to sum to 1, producing a probability distribution for the next word/character in the sequence.

3.2.4 Training the Neural Network

- During training, the actual next word/character (the "label") is known.
- This label's probability (as output by the network) is plucked from the softmax distribution.
- The training objective is to maximize the log likelihood of the correct labels.
- All network parameters (weights and biases of hidden and output layers, and the embedding lookup table C) are optimized using 'backpropagation'.

3.3 Character-Level MLP Implementation in PyTorch

We now transition to implementing this model for character-level language modeling using PyTorch, building on the "makemore" project.

3.3.1 Setup and Data Preparation

We begin by importing necessary libraries, loading the name dataset, and creating character-to-integer mappings.

```
import torch
import torch.nn.functional as F # Convention: F for functional
import matplotlib.pyplot as plt # for plotting

# Load names from file
words = open('names.txt', 'r').read().splitlines()

# Build vocabulary and mappings
chars = sorted(list(set(''.join(words))))
stoi = {s:i+1 for i,s in enumerate(chars)}
stoi['.'] = 0 # Special token for start/end of sequence
itos = {i:s for s,i in stoi.items()}
vocab_size = len(itos) # 27 characters
```

Listing 33: Initial Setup

3.3.2 Dataset Creation

We need to compile a dataset of input-label pairs ('x' and 'y') for the neural network. The 'block$_s$*ize*'*hyperparameterdeterminesthecontextlength*(*howmanypreviouscharactersareusedtopredictthe*

```
# block_size: context length: how many characters do we take
    to predict the next one?
block_size = 3 # Taking 3 characters to predict the 4th

def build_dataset(words):
    X, Y = [], []
    for w in words:
        context = [0] * block_size # Start with padded context
            (0 is '.')
```

```
        for ch in w + '.': # Iterate through word characters + end token
            ix = stoi[ch] # Get integer index of current character
            X.append(context) # Add current context to inputs
            Y.append(ix)      # Add current char's index as label
            context = context[1:] + [ix] # Slide the window: remove first, append current char

    X = torch.tensor(X)
    Y = torch.tensor(Y)
    print(X.shape, Y.shape)
    return X, Y

# Split the data into training, development (validation), and test sets
# Training: optimize model parameters
# Development/Validation: tune hyperparameters (e.g., hidden layer size, embedding size)
# Test: sparingly evaluate final model performance
import random
random.seed(42)
random.shuffle(words) # Shuffle words before splitting

n1 = int(0.8*len(words)) # 80% for training
n2 = int(0.9*len(words)) # 10% for dev, 10% for test

Xtr, Ytr = build_dataset(words[:n1])      # Training set
Xdev, Ydev = build_dataset(words[n1:n2])  # Development/Validation set
Xte, Yte = build_dataset(words[n2:])      # Test set
```

Listing 34: Dataset Creation Function

The ‘context‘ array acts as a rolling window, padding with ‘.‘ (token 0) at the beginning. For a ‘block$_size$‘$of 3$, ‘X‘$contains 3 integers$, and ‘Y‘$contains 1 integer$.

3.3.3 Embedding Lookup Table (C)

We define our embedding table ‘C‘. Initially, we might use a small embedding dimension (e.g., 2) for visualization purposes. For our 27 characters, ‘C‘ will be ‘27 x D‘ (where ‘D‘ is the embedding dimension).

```
emb_dim = 10 # Embedding dimension (e.g., 2 for initial visualization, 10 for better performance)
C = torch.randn((vocab_size, emb_dim)) # 27 chars, each embedded into emb_dim space
```

Listing 35: Embedding Table Initialization

Embedding a Single Integer: We can retrieve the embedding for an integer ‘ix‘ by direct indexing: ‘C[ix]‘.

```
print(C[5]) # Retrieves the embedding vector for character
    index 5
```

Equivalence to One-Hot Encoding and Matrix Multiplication: Conceptually, indexing ‘C[ix]‘ is equivalent to creating a one-hot encoded vector for ‘ix‘ and then multiplying it by ‘C‘.

```
# Example of one-hot encoding (for illustration, not practical
    for indexing)
# Pytorch requires input to be a tensor, not int
one_hot_ix = F.one_hot(torch.tensor(5), num_classes=vocab_size
    ).float()
print(one_hot_ix @ C) # This yields the same result as C[5]
```

However, direct indexing ‘C[ix]‘ is significantly faster and more efficient as it avoids creating the large intermediate one-hot vector and performing matrix multiplication.

Embedding Multiple Integers Simultaneously: PyTorch’s indexing is flexible and allows embedding an entire batch of inputs (‘X‘) simultaneously.

```
# Xtr has shape (num_examples, block_size) e.g., (228146, 3)
embeddings = C[Xtr] # Retrieves embeddings for all integers in
    Xtr
# Resulting shape: (num_examples, block_size, emb_dim) e.g.,
    (228146, 3, 10)
print(embeddings.shape)
```

Listing 36: Batch Embedding

3.3.4 Hidden Layer

The hidden layer performs a linear transformation followed by a ‘tanh‘ non-linearity.

- **Weights (‘W1‘):** Dimensions ‘(block$_s$$ize$∗$emb_d$$im$)$x$$hidden_l$$ayer_s$$ize$‘.**Biases (‘b1‘):**$Dimensions$‘$hidden_l$

```
hidden_layer_size = 200 # Hyperparameter: number of neurons in
    the hidden layer
W1 = torch.randn((block_size * emb_dim, hidden_layer_size))
b1 = torch.randn(hidden_layer_size)
```

Listing 37: Hidden Layer Parameter Initialization

Reshaping Embeddings for Matrix Multiplication: The ‘embeddings‘ tensor has a shape like ‘(batch$_s$$ize$, $block_s$$ize$, $emb_d$$im$)‘.$Toperformmatrixmultiplicationwith$‘$W1$‘($whichexpectsa2Dinp$

- **Naive Concatenation (‘torch.cat‘):** One way is to explicitly slice and concatenate: ‘torch.cat([embeddings[:, 0, :], embeddings[:, 1, :], embeddings[:, 2, :]], dim=1)‘. Using ‘torch.unbind(embeddings, dim=1)‘ provides a general way to get the slices as a tuple, which can then be concatenated. However, ‘torch.cat‘ creates a *new tensor* in memory, making it less efficient.

- **Efficient Reshaping (‘.view()‘):** The most efficient way in PyTorch is to use the ‘.view()‘ method. This operation is extremely efficient because it doesn’t copy or change memory; instead, it manipulates internal tensor attributes (like ‘stride‘ and ‘shape‘) to interpret the underlying one-dimensional memory storage differently.

```
# embeddings.shape: (batch_size, block_size, emb_dim)
# We want to reshape to (batch_size, block_size * emb_dim)
input_to_hidden = embeddings.view(-1, block_size * emb_dim)
# Using -1 lets PyTorch infer the first dimension (batch_size)
# This achieves the desired "concatenation" logically
print(input_to_hidden.shape) # e.g., (228146, 30)
```

Listing 38: Efficient Embedding Reshaping with .view()

Forward Pass through Hidden Layer:

```
# Linear transformation: matrix multiplication and bias
    addition
h_pre_activation = input_to_hidden @ W1 + b1

# Apply tanh non-linearity
h = torch.tanh(h_pre_activation)
print(h.shape) # e.g., (228146, 200)
```

Listing 39: Hidden Layer Computation

Note on Broadcasting: When ‘b1‘ (shape ‘(hidden$_{l}ayer_{s}ize,$)‘)$isaddedto‘h_{p}re_{a}ctivation‘(shape‘(batch$ $wiseadditioncorrectly.$

3.3.5 Output Layer and Logits

The output layer maps the hidden layer activations to logits for each character in the vocabulary.

- **Weights (‘W2‘):** Dimensions ‘hidden$_{l}ayer_{s}izexvocab_{s}ize$‘.**Biases (‘b2‘):**$Dimensions‘vocab_{s}ize$‘.

```
W2 = torch.randn((hidden_layer_size, vocab_size))
b2 = torch.randn(vocab_size)

# Logits calculation
logits = h @ W2 + b2
print(logits.shape) # e.g., (228146, 27)
```

Listing 40: Output Layer Parameter Initialization and Logit Calculation

3.3.6 Loss Function

The ‘logits‘ represent unnormalized scores for each possible next character. To get probabilities, they are typically exponentiated and then normalized (softmax).

```
# Manual calculation:
# counts = logits.exp() # Exponentiate logits to get "fake
    counts"
```

```
# probs = counts / counts.sum(1, keepdim=True) # Normalize to probabilities
# print(probs.shape) # (num_examples, vocab_size), each row sums to 1

# # Get probabilities for the correct characters
# correct_char_probs = probs[range(Xtr.shape[0]), Ytr]
# # Negative Log Likelihood Loss
# loss = -correct_char_probs.log().mean()
# print(loss)
```

Listing 41: Manual Probability and NLL Loss Calculation (for illustration)

Preferring 'F.cross$_e$ntropy' : While the manual calculation works, PyTorch provides 'torch.nn.functional.cross$_e$ntropy'(*oftenaliasedas'F.cross$_e$ntropy'*), *whichisthepreferredwaytocomputethi*

There are several strong reasons to use 'F.cross$_e$ntropy' :

- **Efficiency:** It avoids creating large intermediate tensors (like 'counts' and 'probs') in memory. PyTorch can optimize these clustered operations using "fused kernels," leading to much faster computation.

  Simpler Backward Pass: The analytical derivative for cross-entropy loss is mathematically simpler than backpropagating through individual 'exp', 'sum', and 'log' operations. This leads to a more efficient and robust backward pass implementation.

  Numerical Stability: Cross-entropy is designed to be numerically well-behaved, especially when logits take on extreme values.

  - When logits are very large positive numbers (e.g., 100), 'exp(100)' can lead to floating-point overflow ('inf') and subsequently Not-a-Number ('NaN') results.
- 'F.cross$_e$ntropy'*internallyhandlesthisbysubtractingthemaximumlogitvaluefromalllogitsbeforeexpo*

```
# Using PyTorch's F.cross_entropy (recommended)
# This function internally performs softmax and then negative log likelihood.
# It expects raw logits and target indices (Ytr).
loss = F.cross_entropy(logits, Ytr)
print(loss)
```

Listing 42: Loss Calculation with F.cross$_e$ntropy

3.3.7 Training Loop

The training process involves an iterative loop of forward pass, backward pass (gradient calculation), and parameter updates.

```
# Collect all parameters that require gradients
parameters = [C, W1, b1, W2, b2]
for p in parameters:
    p.requires_grad = True # Enable gradient computation for these tensors

# Initial number of parameters
```

```
num_parameters = sum(p.nelement() for p in parameters)
print(f"Total parameters: {num_parameters}") # e.g., 3400
    for emb_dim=2, hidden_layer_size=100
```

Listing 43: Parameter Collection and Initialization

Mini-Batch Training: To handle large datasets efficiently, we use 'mini-batching'. Instead of calculating gradients over the entire dataset (which is slow), we randomly select a small subset (a mini-batch) for each forward and backward pass.

```
max_steps = 200000 # Number of training iterations
batch_size = 32    # Number of examples in each mini-batch
learning_rate = 0.1 # Initial learning rate (will decay)

for i in range(max_steps):
    # Construct mini-batch
    # Select random indices for the current mini-batch
    ix = torch.randint(0, Xtr.shape[0], (batch_size,)) # (
        batch_size,) tensor of random indices

    # Forward pass on the mini-batch
    emb = C[Xtr[ix]] # (batch_size, block_size, emb_dim)
    h = torch.tanh(emb.view(-1, block_size * emb_dim) @ W1
        + b1) # (batch_size, hidden_layer_size)
    logits = h @ W2 + b2 # (batch_size, vocab_size)
    loss = F.cross_entropy(logits, Ytr[ix]) # Loss for
        this mini-batch

    # Backward pass: zero gradients, compute new gradients
    for p in parameters:
        p.grad = None # Set gradients to zero
    loss.backward() # Computes gradients for all
        parameters that require_grad

    # Parameter update
    for p in parameters:
        p.data -= learning_rate * p.grad # Nudge
            parameters in direction of negative gradient

    # Learning rate decay (example)
    if i == 100000: # After 100,000 steps, reduce LR
        learning_rate = 0.01

    # Optional: print loss periodically
    # if i % 10000 == 0:
    #     print(f"Step {i}: Loss = {loss.item():.4f}")
```

Listing 44: Training Loop with Mini-Batching

3.3.8 Evaluating Performance and Hyperparameter Tuning

Loss on Splits: After training, we evaluate the loss on the entire training set ('Xtr', 'Ytr') and the development set ('Xdev', 'Ydev'). The test set ('Xte', 'Yte')

is reserved for a single final evaluation after all hyperparameter tuning is complete, to avoid overfitting to the test set.

```
@torch.no_grad() # Disable gradient tracking for evaluation
def evaluate_loss(X, Y, C, W1, b1, W2, b2, block_size):
    emb = C[X]
    h = torch.tanh(emb.view(-1, block_size * emb_dim) @ W1 + b1)
    logits = h @ W2 + b2
    loss = F.cross_entropy(logits, Y)
    return loss.item()

train_loss = evaluate_loss(Xtr, Ytr, C, W1, b1, W2, b2, block_size)
dev_loss = evaluate_loss(Xdev, Ydev, C, W1, b1, W2, b2, block_size)
print(f"Final training loss: {train_loss:.4f}")
print(f"Final development loss: {dev_loss:.4f}")
```

Listing 45: Evaluating Loss on Data Splits

Detecting Overfitting and Underfitting:

- If ‘train$_l$oss‘≈ ‘dev$_l$oss‘, $themodelislikely‘underfitting‘.Thismeansthemodelisnotpowerfulenoughto$

Hyperparameter Tuning Example: Scaling Model Capacity Initially, we might see ‘train$_l$oss‘$and‘dev_l$oss‘$aresimilar(e.g., around2.45), indicatingunderfittingcomparedtothel$

1. **Increasing Hidden Layer Size:** Bumping ‘hidden$_l ayer_s ize$‘$(e.g., from100to300neurons)incre$

2. **Visualizing 2D Embeddings (Pre-scaling):** Before increasing embedding dimension, we can visualize the 2D embeddings ‘C‘ to see what the network has learned.

```
# Requires emb_dim = 2 to visualize
plt.figure(figsize=(8,8))
plt.scatter(C[:,0].data, C[:,1].data, s=200) # Plot x, y coordinates from 2D embeddings
for i in range(C.shape[0]):
    plt.text(C[i,0].item(), C[i,1].item(), itos[i], ha="center", va="center", color='white')
plt.grid('minor')
plt.show()
```

Listing 46: Visualizing Character Embeddings (for emb$_d im$

Observation: The network learns meaningful structure. For example, vowels (a, e, i, o, u) often cluster together, suggesting the network treats them as similar or interchangeable. Special characters like ‘.‘ and less common letters like ‘q‘ might be outliers, indicating unique embeddings.

3. **Increasing Embedding Dimension:** If increasing the hidden layer size doesn't sufficiently improve performance, the ‘emb$_d im‘mightbeabottleneck.Increasing‘emb_d im‘(e.g., from2to10)givesth emb$_d im‘becomes‘3 * 10 = 30‘).$

Through such tuning, a significantly lower loss can be achieved (e.g., 'dev$_l oss$'of2.17)$compared to the bigramm

Learning Rate Determination Strategy: Finding an effective 'learning$_r ate$'$is crucial. A common strateg

Initialize parameters.

Sweep 'learning$_r ate$'$logarithmically across a wide range$(e.g., 10^{-4} to 1).

For each learning rate, take a few optimization steps (e.g., 100 or 1000) and record the resulting loss.

Plot 'loss' vs. 'log(learning$_r ate$)'.$The ideal learning rate is typically found in the "valley" of this plot

3.3.9 Sampling from the Model

After training, we can generate new sequences by sampling from the model's predicted probability distribution.

```
# Generate 20 samples
for _ in range(20):
    out = [] # List to store generated characters
    context = [0] * block_size # Start with initial context (all '.')

    while True:
        # Forward pass to get logits for the current context
        emb = C[torch.tensor([context])] # (1, block_size, emb_dim) - single example
        h = torch.tanh(emb.view(1, -1) @ W1 + b1) # (1, hidden_layer_size)
        logits = h @ W2 + b2 # (1, vocab_size)

        # Calculate probabilities using F.softmax (numerically stable)
        probs = F.softmax(logits, dim=1)

        # Sample the next character from the probability distribution
        # torch.multinomial samples indices based on multinomial distribution
        next_char_ix = torch.multinomial(probs, num_samples=1).item()

        # Update context window and record the new character
        context = context[1:] + [next_char_ix] # Slide window
        out.append(next_char_ix)

        # Break if we generate the end-of-sequence token ('.')
        if next_char_ix == 0:
            break
```

```
26
27      # Decode and print the generated name
28      print(''.join(itos[ix] for ix in out))
```

Listing 47: Generating Samples from the Trained Model

The generated samples will now appear much more "name-like" than those from the bigram model, indicating significant progress.

3.4 Further Improvements and Exploration

The model's performance can be further enhanced by tuning various hyperparameters and exploring advanced techniques:

- **Model Architecture:**
  - Number of neurons in the hidden layer ('hidden$_layer_size$').$Dimensionalityoftheembe$
- Number of characters in the input context ('block$_size$').
- **Optimization Details:**
  - Total number of training steps.
  - Learning rate schedule (how it changes over time, e.g., decay strategies).
  - Batch size (influences gradient noise and convergence speed).
- **Reading the Paper:** The original Bengio et al. (2003) paper contains additional ideas for improvements.

3.5 Google Colab Accessibility

For ease of experimentation, the Jupyter notebook for this lecture is available via Google Colab. This allows you to run and modify the code directly in your browser without any local installation of PyTorch or Jupyter. The link is typically provided in the video description.

4 Lecture 4: Building makemore Part 3 - Activations, Gradients, & BatchNorm

Abstract

Welcome, aspiring deep learning practitioners! In this comprehensive set of lecture notes, we delve deeper into the intricate world of neural networks, specifically focusing on the critical aspects of activation functions, gradient flow during backpropagation, and the transformative technique of Batch Normalization. Our journey continues from the foundational Multilayer Perceptron (MLP) for character-level language modeling, as implemented in the previous session, moving towards more complex architectures like Recurrent Neural Networks (RNNs) and Transformers. Before we tackle those, however, a solid intuitive understanding of what happens inside an MLP during training is paramount. This deep dive

into activations and gradients is crucial for comprehending the historical development of neural network architectures and why certain innovations were necessary to enable the training of deeper, more expressive models.

4.1 Starting Point: The MLP for makemore

Our initial setup is largely based on the previous MLP code, now refactored for clarity. We are still building a character-level language model that takes a few past characters as input to predict the next character in a sequence.

4.1.1 Code Refinements

- We've removed "magic numbers" by externalizing hyperparameters like embedding space dimensionality ('n$_e$*mbd*')*andnumberofhiddenunits*('n_h*idden*').*Thismake*

- Code has been refactored for better readability, including more comments and organized evaluation logic for different data splits (train, validation, test).

4.1.2 The `torch.no`$_g$*radContext*

An important optimization used during evaluation is the 'torch.no$_g$*rad*'*decoratororcontextmanag*

```
@torch.no_grad()
def evaluate_split(split_name):
    # ... computation where gradients are not needed
        ...
```

4.1.3 Current Model Performance

The model currently generates much nicer-looking words compared to a simpler Bigram model, though it's still not perfect. The typical train and validation loss observed was around 2.16.

4.2 Scrutinizing Initialization: Why It Matters So Much

The first issue we observe is a significantly high initial loss, indicating improper network configuration from the start.

4.2.1 Problem 1: Overconfident, Incorrect Predictions in the Output Layer

- **Observation**: At iteration 0, the loss is approximately 27, which rapidly decreases.
- **Expected Initial Loss**: For a 27-character vocabulary (including a special '.' token), if the network were truly uninformed, it should output a uniform probability distribution for the next character. The probability for any character would be $1/27$. The negative log-likelihood loss for a uniform distribution over 27 classes is $-\log(1/27) \approx 3.29$.

- **Discrepancy**: A loss of 27 is drastically higher than the expected 3.29. This implies the neural network is "confidently wrong" at initialization, assigning very high probabilities to incorrect characters.
- **Root Cause**: The 'logits' (raw outputs before softmax) are taking on extreme values, which, after softmax, lead to highly peaked (confident) but incorrect probability distributions.

4.2.2 Solution 1: Normalizing Output Logits

To fix this, we want the logits to be roughly zero (or equal) at initialization, yielding a uniform probability distribution after softmax.

- **Bias Initialization**: The output bias 'B2' is currently initialized randomly. We want it to be approximately zero to avoid arbitrary offsets.
- **Weight Scaling**: The logits are computed as 'H @ W2 + B2'. To make logits small, we should scale down 'W2'.
- **Why not exact zero weights?**: While setting 'W2' to exactly zero would give the perfectly uniform distribution we desire at initialization, it's generally avoided to maintain symmetry breaking. Small random values allow different neurons to learn different features from the start. We choose a small scaling factor like '0.01'.

Before Fix: Initial loss ≈ 27.0. **After Fix**: Initial loss ≈ 3.32 (closer to expected 3.29).

```
# Original initialization (simplified):
# self.W2 = torch.randn((n_hidden, vocab_size))
# self.B2 = torch.randn(vocab_size)

# Fixed initialization:
self.W2 = torch.randn((n_hidden, vocab_size)) * 0.01 # Scale down W2
self.B2 = torch.zeros(vocab_size) # Initialize B2 to zeros
```

4.2.3 Impact of Fixing Logit Initialization

- The loss plot no longer exhibits a "hockey stick" appearance (a sharp initial drop followed by a plateau). This is because the initial easy task of "squashing down the logits" is removed, allowing the network to immediately focus on meaningful learning.
- The final validation loss improves slightly (e.g., from 2.17 to 2.13). This is due to more productive training cycles, as the network isn't wasting early iterations on fixing a bad initialization.

4.3 Problem 2: Saturation of Hidden Layer Activations (Tanh)

Even with a reasonable initial loss, there's a deeper problem within the hidden layer activations, particularly when using squashing activation functions like Tanh.

4.3.1 Observation: Saturated Tanh Activations

- The histogram of ‘H‘ (hidden state activations after Tanh) shows most values are concentrated at -1 or 1.
- This means the Tanh neurons are ”saturated,” operating in the flat regions of the Tanh curve.
- The ‘pre-activations‘ (inputs to Tanh) are very broad, ranging from -5 to 15, causing this saturation.

4.3.2 The Vanishing Gradient Problem with Tanh

- **Tanh Backward Pass**: Recall the derivative of Tanh: ‘d(tanh(x))/dx = 1 - tanh(x)^2‘.$In backpropagation, the local gradient for Tanh is ‘1-T^2‘, where ‘T‘ is the outp$
  $If ‘T‘ is close to -1 or 1 (i.e., the neuron is saturated), ‘1-T^2‘ becomes very close to zero. This loc$

- **Intuition**: When an activation is in a flat region, changing its input has little to no impact on its output, and thus no impact on the loss. Consequently, its associated weights and biases receive negligible gradients and cannot learn.
- **Dead Neurons**: If a neuron consistently lands in the saturated region for all training examples (i.e., its column of activations is entirely ”white” when visualizing ‘abs(H) ¿ 0.99‘), it becomes a ”dead neuron” that never learns.

4.3.3 Other Nonlinearities and Dead Neurons

- **Sigmoid**: Suffers from the exact same vanishing gradient issue as Tanh due to its squashing nature.
- **ReLU**: Has a flat region for negative inputs, where the gradient is exactly zero. A ”dead ReLU neuron” occurs if its pre-activation is always negative, causing it to never activate and its weights/bias to never learn. This can happen at initialization or during training with a high learning rate.
- **Leaky ReLU**: Designed to mitigate this by having a small non-zero slope for negative inputs, ensuring gradients almost always flow.
- **ELU**: Also has flat parts for negative values, potentially suffering from similar issues.

4.3.4 Solution 2: Normalizing Hidden Layer Activations

We want the ‘pre-activations‘ feeding into Tanh to be closer to zero, so that the Tanh outputs are more centered around zero and less saturated.

- **Bias Initialization**: Similar to ‘B2‘, ‘B1‘ (hidden layer bias) can be initialized to small numbers (e.g., multiplied by 0.01) to introduce a little entropy and diversity, aiding optimization.
- **Weight Scaling**: Scale down ‘W1‘ (hidden weights). Through experimentation, a factor like ‘0.2‘ or ‘0.1‘ works well initially.

Before Fix: ‘W1 = torch.randn((n$_e$*mbd* ∗ *block$_s$ize*, *n$_h$idden*))‘**Fixed initialization** : ‘$W1 = torch.randn((n_embd * block_size, n_hidden)) * 0.2$‘

```
# Original initialization (simplified):
# self.W1 = torch.randn((n_embd * block_size, n_hidden))
# self.B1 = torch.randn(n_hidden)

# Fixed initialization:
self.W1 = torch.randn((n_embd * block_size, n_hidden)) * 0.2 # Scale down W1
self.B1 = torch.randn(n_hidden) * 0.01 # Initialize B1 with small entropy
```

4.3.5 Impact of Fixing Tanh Saturation

- The histogram of ‘H‘ shows a much better distribution, with ‘pre-activations‘ between -1.5 and 1.5. There are no saturated neurons (no ”white” in the visualization).
- The final validation loss further improves (e.g., from 2.13 to 2.10).
- This illustrates that better initialization leads to more productive training and ultimately better performance, especially crucial for deeper networks where these problems compound.

4.4 Principled Initialization: Beyond ”Magic Numbers”

Manually tuning these scaling factors (‘0.1‘, ‘0.2‘, ‘0.01‘) becomes impossible for deep networks with many layers. We need a principled approach.

4.4.1 The Goal: Preserving Activation Distributions

The objective is to ensure that activations throughout the neural network maintain a relatively similar distribution, ideally a unit Gaussian (mean 0, standard deviation 1). If activations grow too large, they saturate; if too small, they vanish.

4.4.2 Mathematical Derivation for Linear Layers

Consider a linear layer ‘Y = X @ W‘. If ‘X‘ and ‘W‘ are sampled from zero-mean Gaussian distributions, the standard deviation of ‘Y‘ is given by: $\sigma_Y = \sigma_X \cdot \sigma_W \cdot \sqrt{\text{fan\_in}}$.

To maintain $\sigma_Y = \sigma_X = 1$ (i.e., unit Gaussian activations), we need to set $\sigma_W = 1/\sqrt{\text{fan\_in}}$, where ‘fan$_i$n‘$is the number of input features to the weight matrix$.

```
# Example demonstrating standard deviation preservation
# x: input (1000 examples, 10 dimensions)
# W: weights (10 inputs, 200 outputs)
# fan_in = 10 (number of inputs to each neuron in W)

```

```
# When W is scaled by 1/sqrt(fan_in):
# W = torch.randn((10, 200)) / (10**0.5)
# y = x @ W
# y.std() will be approximately 1.0
```

4.4.3 Kaiming Initialization (He Initialization)

- **Context**: The paper "Delving Deep into Rectifiers" by Kaiming He et al. (2015) extensively studied initialization, particularly for Convolutional Neural Networks (CNNs) and ReLU/PReLU nonlinearities.
- **The Gain Factor**: For ReLU, which clamps negative values to zero (discarding half the distribution), an additional gain factor is needed. They found that weights should be initialized with a zero-mean Gaussian distribution with standard deviation $\sigma_W = \sqrt{2/\text{fan\_in}}$. The '$\sqrt{2}$' compensates for the "loss" of half the distribution.
- **PyTorch Implementation**: 'torch.nn.init.kaiming$_n ormal \cdot implementsthis.Ittakes`mode$

  - 'linear' (or identity): Gain is 1. $\sigma_W = \sqrt{1/\text{fan\_in}}$.
  - 'relu': Gain is $\sqrt{2}$. $\sigma_W = \sqrt{2/\text{fan\_in}}$.
  - 'tanh': Advised gain is 5/3. This is because Tanh is a "contractive transformation" (it squashes the tails), so a gain is needed to "fight the squeezing" and renormalize the distribution.

4.4.4 Practical Initialization with Kaiming

For our Tanh-based MLP, the 'fan$_i n$'for'$W1$'is'$n_embd * block_size$'(30$inourcase$).$TheKaiminginitialization$
$(5/3)/sqrt(fan_in)$'.

```
# Initializing W1 using Kaiming-like approach for Tanh
fan_in_W1 = n_embd * block_size # Which is 30
gain_tanh = (5/3) # Kaiming gain for Tanh
std_W1 = gain_tanh / (fan_in_W1**0.5)

self.W1 = torch.randn((n_embd * block_size, n_hidden))
    * std_W1
```

This principled initialization leads to results comparable to our manual tuning but is scalable to much deeper networks without guesswork.

4.4.5 Modern Innovations Reducing Initialization Sensitivity

While proper initialization is beneficial, its precise calibration has become less critical due to several modern innovations:

- **Residual Connections**: These allow gradients to bypass layers, preventing vanishing/exploding gradients in deep networks.

- **Normalization Layers**: Techniques like Batch Normalization, Layer Normalization, and Group Normalization actively control activation statistics during training.
- **Better Optimizers**: Advanced optimizers like RMSprop and Adam adapt learning rates for different parameters, making training more robust to initialization.

4.5 Batch Normalization: A Game Changer

Introduced by Google in 2015, Batch Normalization (BatchNorm) fundamentally changed the landscape of training very deep neural networks.

4.5.1 The Core Idea: Standardizing Activations

- Instead of carefully initializing weights to ensure Gaussian activations, BatchNorm simply *forces* them to be Gaussian-like (zero mean, unit standard deviation) by normalizing them.
- This standardization is a fully differentiable operation, allowing it to be integrated into the network and trained via backpropagation.

- The normalization is performed *per batch* and *per neuron*. For a pre-activation tensor $`H_preact` of shape (batch_size, num_neurons), the mean and standard deviation are calculated across the batch_s
  The normalization formula for an activation x_i within a mini-batch is: $\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$ where μ_B is the mini-batch mean, σ_B^2 is the mini-batch variance, and ϵ is a small constant to prevent division by zero.

4.5.2 Learnable Scale and Shift Parameters (Gamma and Beta)

- While normalizing to unit Gaussian is good for initialization, we don't want to *force* activations to always be unit Gaussian throughout training. The network should have the flexibility to adjust the distribution as needed.
- BatchNorm introduces two learnable parameters per neuron: 'gain' (gamma, γ) and 'bias' (beta, β).
- The final output of a BatchNorm layer is: $y_i = \gamma \hat{x}_i + \beta$.
- Initially, γ is set to 1 and β to 0, ensuring unit Gaussian output. During training, γ and β are updated via backpropagation, allowing the network to scale and shift the normalized activations if optimal for learning.
- These γ and β parameters are part of the network's trainable parameters and contribute to the overall parameter count.

```
# Example of batch normalization in forward pass
# H_preact: (batch_size, n_hidden)
mean = H_preact.mean(0, keepdim=True) # mean across batch dimension, (1, n_hidden)
std = H_preact.std(0, keepdim=True)   # std across batch dimension, (1, n_hidden)
```

```
5
6  H_preact_norm = (H_preact - mean) / (std + 1e-5) # Add
       epsilon to prevent div by zero
7
8  # Learnable parameters (initialized at 1 for gain, 0
       for bias)
9  bn_gain = torch.ones((1, n_hidden))
10 bn_bias = torch.zeros((1, n_hidden))
11
12 H = bn_gain * H_preact_norm + bn_bias # Scaled and
       shifted normalized activations
```

4.5.3 Placement of BatchNorm Layers

It is customary to place BatchNorm layers right after linear or convolutional layers and before the nonlinearity (e.g., Tanh or ReLU). This sequence helps control the scale of activations at every point in the neural network, simplifying the tuning of weight matrices.

4.5.4 BatchNorm's Side Effects: Regularization and Coupled Examples

- **Coupled Examples**: BatchNorm couples examples within a batch. The normalization for any given input depends on the statistics (mean and standard deviation) of all other examples in the current batch. This introduces a "jitter" or noise into the activations.
- **Regularization**: This "jittering" acts as a form of regularization, akin to data augmentation. It makes it harder for the network to overfit to specific examples, thus improving generalization. This secondary regularizing effect is a key reason BatchNorm has been so difficult to replace despite its drawbacks.

4.5.5 BatchNorm at Inference Time

- **Problem**: During inference, we typically process single examples or very small batches. Using batch statistics would lead to unstable and non-deterministic predictions.
- **Solution: Running Statistics**: BatchNorm layers maintain "running mean" and "running variance" (or standard deviation) during training using an exponential moving average (EMA). These are not updated by gradient descent but are buffers updated "on the side" during the forward pass.
- **Inference Behavior**: At test time, these fixed running statistics are used instead of mini-batch statistics to normalize inputs, ensuring deterministic and consistent predictions for individual examples.

```
1 # Updating running mean and std during training
2 # (bn_mean_running and bn_std_running are initialized
      as buffers)
```

```
with torch.no_grad():
    bn_mean_running = 0.999 * bn_mean_running + 0.001
       * mean # (0.001 is momentum)
    bn_std_running = 0.999 * bn_std_running + 0.001 *
       std

# At inference, use:
# H_preact_norm = (H_preact - bn_mean_running) / (
   bn_std_running + 1e-5)
```

4.5.6 Bias Elimination in Preceding Layers

When a BatchNorm layer follows a linear or convolutional layer, the bias parameter (‘B1‘ in our MLP’s ‘W1 @ H + B1‘) in that preceding layer becomes redundant. BatchNorm calculates and subtracts the mean, effectively cancelling out any constant bias added before it. Therefore, if a layer is followed by BatchNorm, its ‘bias‘ parameter can be set to ‘False‘ in PyTorch’s ‘nn.Linear‘ or ‘nn.Conv2d‘ to avoid unnecessary parameters.

```
# In a BatchNorm-enabled network, for a linear layer
   followed by BatchNorm:
# self.linear = nn.Linear(input_features,
   output_features, bias=False)
# self.bn = nn.BatchNorm1d(output_features)
```

4.5.7 BatchNorm in PyTorch (`torch.nn.BatchNorm1d`)

- **Parameters**: Takes ‘num$_features$‘($dimensionality of the activation, e.g.,$ ‘n_hidden‘ = 200).**Keyword Arguments** :
-
  - ‘eps‘ (epsilon): Default ‘1e-5‘. Prevents division by zero.
  - ‘momentum‘: Default ‘0.1‘. Controls the EMA update rate for running statistics. Smaller momentum (‘0.001‘) is better for small batch sizes where batch statistics fluctuate more.
  - ‘affine‘: Default ‘True‘. Determines if learnable ‘gamma‘ (weight) and ‘beta‘ (bias) parameters are included.

- ‘track$_running_stats$‘ : $Default ‘True‘. Determines if running mean/variance are tracked. Set to ‘False‘ if you p$

Internal State: ‘weight‘ (gamma) and ‘bias‘ (beta) are ‘parameters‘ (trainable via backprop). ‘running$_mean$‘and‘$running_var$‘are‘$buffers$‘($updated by EMA, not backprop$).**‘.training‘ attribute** : $This flag (default ‘True‘) dictates behavior. If ‘True‘, batch statistics are used, and running stats are updated. I$

4.5.8 The BatchNorm ”Motif” in Deep Networks

The common pattern for deep neural networks, especially those using ReLU or Tanh, is a sequence like: **Linear/Convolutional Layer → BatchNorm Layer → Non-linearity (e.g., Tanh/ReLU)**.

This motif (e.g., ‘Conv -¿ BatchNorm -¿ ReLU‘) is prevalent in architectures like ResNet, stabilizing training by continuously controlling activation distributions.

4.6 Diagnostic Tools for Neural Network Health

Understanding the internal state of a neural network during training is crucial. We can visualize histograms and statistics of activations, gradients, and parameters.

4.6.1 Forward Pass Activations

- **What to Look For**: Histograms of activations (e.g., after Tanh layer). We track mean, standard deviation, and "percent saturation" (e.g., 'abs(T) ¿ 0.97' for Tanh).
- **Desired State**: Homogeneous distributions across layers, roughly centered around zero, with reasonable standard deviation, and low saturation (e.g., $\approx 5\%$) for squashing non-linearities like Tanh.
- **Issue Indicators**:
  - *Shrinking to zero*: Activations become very small, leading to vanishing gradients.
  - *Exploding*: Activations become very large, leading to saturation and vanishing gradients.

4.6.2 Backward Pass Gradients

- **What to Look For**: Histograms of gradients of activations ('.grad' attribute of the 'out' tensors after a backward pass). We track mean and standard deviation.
- **Desired State**: Homogeneous distributions across layers, indicating gradients flow effectively without shrinking or exploding.
- **Issue Indicators**: Vanishing or exploding gradients, where gradients in early layers are significantly smaller or larger than those in later layers.

4.6.3 Parameter Distributions

- **What to Look For**: Histograms of the weights (parameters) themselves. We track mean and standard deviation.
- **Desired State**: Parameters should have reasonable means and standard deviations, ideally maintaining a good spread.
- **Issue Indicators**: Very skewed distributions, or large discrepancies in magnitude across different layers.

4.6.4 Update-to-Data Ratio

- **What to Look For**: This is a crucial diagnostic. It's the ratio of the magnitude of the update applied to a parameter ('learning$_rate$ $*$ $grad$')$tothemagnitudeoftheparameteritself('param_data').Specifically,'std(learning_rate$ $*$ $param.grad)/std(param.data)'.Oftenvisualizedona'log10'scale.$**Intuition** : $Thisratioindicateshowmuchaparameterchangesrelativetoitscurrentvalueinoneupdatest$

- **Desired State**: A rough heuristic is that ‘log10(ratio)‘ should be around -3. This means the updates are roughly 1/1000th the magnitude of the parameters, allowing for stable, continuous learning.
- **Issue Indicators**:
  - *Too High (e.g., ¿ -2)*: The learning rate is likely too high, or updates are too large, causing parameters to thrash or overshoot.
  - *Too Low (e.g., ¡ -4)*: The learning rate is likely too low, or updates are too small, causing parameters to learn too slowly.
  - *Discrepancies across layers*: If some layers have significantly different update ratios, it suggests an imbalance in training speed across the network.

4.7 Building Custom PyTorch-like Modules

To facilitate structured neural network construction and diagnostics, we’ve refactored our code into custom modules that mimic PyTorch’s ‘nn.Module‘ API.

4.7.1 `Linear` Module

- Initializes ‘weight‘ and optional ‘bias‘ tensors.
- ‘weight‘ is initialized using Kaiming-like initialization (‘1 / sqrt(fan$_in$)‘$fordefault, withgain$
B‘.

- ‘parameters‘ method returns trainable tensors (‘weight‘, ‘bias‘).

4.7.2 `BatchNorm1d` Module

- Initializes ‘gamma‘ (weight) and ‘beta‘ (bias) as trainable parameters.
- Initializes ‘running$_mean$‘and‘$running_variance$‘$asnon-trainablebuffers.Includesan$‘$epsil$

- Has a ‘.training‘ attribute to switch between training (batch stats, update running stats) and evaluation (running stats, no updates) modes.
- ‘forward‘ method calculates batch mean/variance (if training) or uses running mean/variance (if evaluating), normalizes, then applies gamma and beta.
- Crucially, the running mean/variance update is wrapped in ‘torch.no$_grad$()‘$topreventbuildingacomputationa$
$backpropagatedbuffers$.

4.7.3 `Tanh` Module

- A simple module that wraps ‘torch.tanh‘.
- Has no trainable parameters.

These modular components (layers) can then be easily stacked into a sequential model, resembling how models are constructed in ‘torch.nn‘.

4.8 Conclusion

- This lecture emphasized the critical importance of understanding the internal dynamics of neural networks: activations, gradients, and their distributions.
- We learned how improper initialization can lead to high initial loss and saturated activations, hindering effective training.
- Principled initialization methods, like Kaiming initialization, provide a systematic way to scale weights to preserve activation distributions.
- Batch Normalization was introduced as a pivotal innovation that stabilizes deep network training by actively normalizing activations. It simplifies the initialization challenge and acts as a regularizer, though it introduces complexity with coupled examples and running statistics.
- We explored powerful diagnostic tools (histograms of activations, gradients, parameters, and the update-to-data ratio) that help assess the "health" of a neural network during training, guiding hyperparameter tuning like learning rate selection.
- While these advancements have significantly improved training reliability, the field of initialization and optimization remains an active research area, and there are still open questions.
- Our current model's performance may now be bottlenecked by architectural limitations (e.g., context length), pointing towards the need for more advanced architectures like RNNs and Transformers, which we will explore in future lectures.

These principles and diagnostic techniques will become even more vital as we transition to deeper and more complex neural network architectures in the future.