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.

```python
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).

```python
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), '+'
            )

        def _backward():
            # For addition, the gradient is simply passed through (local derivative is 1)
            self.grad += out.grad * 1.0  # Accumulate gradient
            other.grad += out.grad * 1.0  # Accumulate gradient
        out._backward = _backward
        return out

    def __mul__(self, other):
        # Handle scalar multiplication (e.g., Value * 2)
        other = other if isinstance(other, Value) else Value(other)
        out = Value(self.data * other.data, (self, other), '*')

        def _backward():
            # For multiplication, local derivatives are the 'other' operand
            self.grad += out.grad * other.data  # Accumulate gradient
            other.grad += out.grad * self.data  # Accumulate gradient
        out._backward = _backward
        return out

    # For operations like 2 * a (reversed multiplication)
    def __rmul__(self, other):
        return self * other

    def __pow__(self, other):
        # This implementation expects 'other' to be a scalar (int or float), not a Value object
        assert isinstance(other, (int, float)), "only supporting int/float powers for now"
        out = Value(self.data**other, (self,), f'**{other}')

        def _backward():
            # Power rule: d/dx(x^n) = n*x^(n-1)
            self.grad += out.grad * (other * (self.data**(other-1)))
        out._backward = _backward
        return out

    def __neg__(self):  # -self
        return self * -1

    def __sub__(self, other):  # self - other
        return self + (-other)

    def __truediv__(self, other):  # self / other
        # Division is implemented as multiplication by a negative power
        return self * other**-1

    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.

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

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

Listing 5: Topological Sort Logic (within backward method)

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

1.5 Building a Neural Network (MLP)

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

1.5.1 The Neuron

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

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

```python
import random

class Neuron:
    def __init__(self, nin):
        # nin: number of inputs to this neuron
        self.w = [Value(random.uniform(-1,1)) for _ in range(nin)]
        self.b = Value(random.uniform(-1,1))

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

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

Listing 6: Neuron Class Implementation

1.5.2 The Layer

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

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

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

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

Listing 7: Layer Class Implementation

1.5.3 The Multi-Layer Perceptron (MLP)

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

```python
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])
```

```

# Forward pass to get predictions
ypred = [n(x) for x in xs]
print("Initial predictions:", [p.data for p in ypred])

# Calculate Mean Squared Error loss
loss = sum([(yout - ygt)**2 for ygt, yout in zip(ys, ypred)])
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.

```
# Re-initialize the network for a fresh start
n = MLP(3, [4, 4, 1])

# Training hyperparameters
num_epochs = 50  # Number of training iterations
learning_rate = 0.05

for k in range(num_epochs):
    # 1. Forward pass
    ypred = [n(x) for x in xs]

    # 2. Calculate loss
    loss = sum([(yout - ygt)**2 for ygt, yout in zip(ys, ypred
        )])

    # 3. Zero gradients (VERY IMPORTANT!)
    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.

```python
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,.)‘.

```python
# 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.

```python
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.

```python
# 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:

```python
# 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.

```python
# 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**.

```python
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.

```python
# 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.

```python
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.

```python
    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$_encoded@W‘.Thisoperatio$

```python
    # 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$_encoded‘isone-hot,‘x_encoded@W‘effectively”plucksout”therowof‘W‘correspond$
$hotinputvector.Thismeans‘logits[i]‘(forthei-thexample)willbeidenticalto‘W[ix1]‘,where‘ix1‘istheinteger$

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 $= e^{\text{logits}}$. 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$_grad = True‘(inourcase, ‘W‘).Thesegradientsarethenstoredinthe‘.grad‘attributeof‘W‘$.

```
    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$_rate‘(e.g., 0.1 or 50) controlsthesiz$

```
    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$_r eg`leadstoasmoothermodel, analogoustoaddingmorefakecounts.$

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.

```python
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$_s ize‘of3,‘X‘contains3integers, and‘Y‘contains1integer.$

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$_{s}ize*emb_{d}im)xhidden_{l}ayer_{s}ize$‘.**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$_layer_size,)‘)isaddedto‘h_pre_activation‘(shape‘(batch$
$wiseadditioncorrectly.$

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$_layer_sizexvocab_size$‘.**Biases (‘b2‘):**$Dimensions$‘$vocab_size$‘.

```
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$_entropy$‘ : While the manual calculation works, PyTorch provides ‘torch.nn.functional.cross$_entropy‘(oftenaliasedas‘F.cross_entropy‘), whichisthepreferredwaytocomputethi$
There are several strong reasons to use ‘F.cross$_entropy$‘ :

- **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$_entropy‘internallyhandlesthisbysubtractingthemaximumlogitvaluefromalllogitsbeforeexpo$

```
# 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$_entropy$

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.

```python
@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$_loss$‘≈ ‘dev$_loss$‘$, themodelislikely‘underfitting‘.Thismeansthemodelisnotpowerfulenoughto$

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

1• **Increasing Hidden Layer Size:** Bumping ‘hidden$_layer_size$‘$(e.g., from100to300neurons)incre$

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

```python
# 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$_dim$‘$mightbeabottleneck.Increasing$‘$emb_dim$‘$(e.g., from2to10)givesth$ emb_dim‘$becomes$‘$3 * 10 = 30$‘).

Through such tuning, a significantly lower loss can be achieved (e.g., ‘dev$_l$*oss‘of*2.17)*comparedtothebigramm*

Learning Rate Determination Strategy: Finding an effective ‘learning$_r$*ate‘iscrucial.Acommonstrateg*

Initialize parameters.

Sweep ‘learning$_r$*ate‘logarithmicallyacrossawiderange*(*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*)‘.*Theideallearningrateistypicallyfoundinthe”valley”ofthisplot*

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

    # Decode and print the generated name
    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‘$.Inbackpropagation,thelocalgradientforTanhis‘1-T^2‘,where‘T‘istheoutp$ $If‘T‘iscloseto-1or1(i.e.,theneuronissaturated),‘1-T^2‘becomesveryclosetozero.Thisloca$
- **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$_emb d * block_size, n_hidden))$‘**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$_n ormal‘implementsthis.Ittakes‘mode$

  - ‘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$_i n‘for‘W1‘is‘n_e mbd*block_s ize‘(30 in our case).The Kaiming initialization$
$(5/3)/sqrt(fan_i n)$‘.

```
# 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$_{p}react`of shape`(batch_{s}ize, num_{n}eurons)`, the mean and standard deviation are calculated across the`batch_{s}$

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)
```

```

H_preact_norm = (H_preact - mean) / (std + 1e-5) # Add
     epsilon to prevent div by zero

# Learnable parameters (initialized at 1 for gain, 0
    for bias)
bn_gain = torch.ones((1, n_hidden))
bn_bias = torch.zeros((1, n_hidden))

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.

```
# Updating running mean and std during training
# (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$_{f}eatures‘(dimensionalityoftheactivation, e.g., ‘n_{h}idden‘ =$ 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$_{r}unning_{s}tats‘ : Default‘True‘.Determinesifrunningmean/variancearetracked.Setto‘False‘ifyoup$

Internal State: ‘weight‘ (gamma) and ‘bias‘ (beta) are ‘parameters‘ (trainable via back-prop). ‘running$_{m}ean‘and‘running_{v}ar‘are‘buffers‘(updatedbyEMA, notbackprop)$.**‘.training‘ attribute** : $Thisflag(default‘True‘)dictatesbehavior.If‘True‘, batchstatisticsareused, andrunningstatsareupdated.I$

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$_{i}n)‘fordefault, withgai$
  B‘.
- ‘parameters‘ method returns trainable tensors (‘weight‘, ‘bias‘).

4.7.2 `BatchNorm1d` Module

- Initializes ‘gamma‘ (weight) and ‘beta‘ (bias) as trainable parameters.
- Initializes ‘running$_{m}ean‘and‘running_{v}ariance‘asnon-trainablebuffers.Includesan‘epsil$
- 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$_{g}rad()‘topreventbuildingacomputationa$
  $backpropagatedbuffers.$

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.

5 Lecture 5: Building makemore Part 4 - Becoming a Backprop Ninja

Abstract

In the journey of implementing the `makemore` neural network, we've successfully built a multi-layer perceptron (MLP) and achieved reasonable loss values. A crucial line of code that orchestrates the learning process is `loss.backward()`, which relies on PyTorch's automatic differentiation engine (autograd) to compute gradients. While convenient, this lecture emphasizes the profound importance of understanding and manually implementing the backward pass at the tensor level, rather than relying solely on the framework's abstraction. We will demystify backpropagation by implementing it step-by-step, from atomic operations to analytical derivations for cross-entropy and batch normalization.

5.1 Introduction: Demystifying Backpropagation

Andrej Karpathy refers to backpropagation as a "leaky abstraction". This means that while it seems to magically make neural networks work, it's not a foolproof mechanism.

A lack of understanding of its internals can lead to significant issues, such as suboptimal performance or outright failures, making debugging a formidable challenge.

Understanding manual backpropagation provides several key benefits:

- **Debugging Capabilities**: It empowers you to debug neural networks effectively. Common issues like dying gradients (due to saturated activations), dead neurons, or exploding/vanishing gradients (especially in recurrent neural networks) require a deep understanding of how gradients are calculated and flow through the network.
- **Avoiding Subtle Bugs**: Unbeknownst to many, even experienced developers can introduce subtle but major bugs if they don't grasp backpropagation. An example includes misinterpreting gradient clipping as loss clipping, which can inadvertently set gradients of outliers to zero, effectively ignoring them.
- **Full Explicitness**: Manually writing the backward pass removes the "magic" and makes every calculation explicit, fostering confidence and clarity in the network's behavior.
- **Enhanced Debugging Skills**: This exercise hones your ability to debug complex systems, translating to stronger overall development skills.
- **Deeper Intuition**: It cultivates a profound intuition about how gradients flow backward from the loss function through all intermediate tensors and ultimately to the network's parameters.

5.1.1 A Historical Perspective

While today, manually writing the backward pass is typically reserved for educational purposes, approximately ten years ago, it was the pervasive standard in deep learning. Developers, including Andrej Karpathy himself, routinely implemented their backward passes by hand. Early libraries, such as those written in Matlab for Restricted Boltzmann Machines around 2010, explicitly managed gradients inline without widespread use of autograd engines. Even in 2014, with the advent of Python and NumPy for deep learning, it was common practice to implement both the forward pass and the backward pass manually, often using gradient checkers to verify numerical and analytical gradient agreement.

5.2 Lecture Setup and Objectives

Our objective is to replace the `loss.backward()` call in our existing two-layer MLP (which includes a batch normalization layer) with a manually implemented backward pass at the tensor level.

5.2.1 MLP Architecture Recap

Our current MLP is structured as:

- Embedding layer for characters.
- First linear layer.

- Batch Normalization layer.
- Tanh activation function.
- Second linear layer.
- Cross-entropy loss function.

The forward pass will remain largely identical, but the backward pass will be entirely manual.

5.2.2 Key Setup Elements

1. **Gradient Comparison Utility (`cmp`)**: A utility function is introduced to compare our manually calculated gradients (`dt`) with those computed by PyTorch's autograd (`t.grad`). It checks for exact and approximate equality (accounting for floating-point inaccuracies) and reports the maximum difference.
2. **Parameter Initialization**: Biases are initialized with small random numbers instead of exact zeros. This is a deliberate choice to prevent masking potential errors in gradient calculations, as zero-initialized variables can sometimes simplify gradient expressions in a way that hides bugs. A spurious bias (`B1`) is also added before batch normalization in the first layer, just to ensure its gradient can be correctly calculated, even though it's typically unnecessary.
3. **Expanded Forward Pass**: The forward pass is broken down into more explicit, manageable chunks with many intermediate tensors (e.g., `logprobs`, `probs`, `counts`). This is essential because we will be calculating derivatives for each of these intermediate tensors in the backward pass.
4. **Gradient Naming Convention**: For every intermediate tensor `X` in the forward pass, its corresponding gradient (derivative of the loss with respect to `X`) will be named `dX` (e.g., `dlogprobs` for `logprobs`).

5.2.3 Exercise Breakdown

The lecture is structured around four main exercises:

1. **Exercise 1: Atomic Backpropagation**: Manually backpropagate through every single atomic operation in the forward pass, step-by-step. This involves calculating all the `dX` variables and verifying them with PyTorch's `autograd` using the `cmp` function.
2. **Exercise 2: Analytical Cross-Entropy Backprop**: Derive and implement a single, efficient analytical formula for the backward pass through the cross-entropy loss, rather than breaking it into atomic operations.
3. **Exercise 3: Analytical Batch Normalization Backprop**: Similarly, derive and implement a single, efficient analytical formula for the backward pass through the entire batch normalization layer.

4. **Exercise 4: Full Manual Training Loop**: Integrate the manual backward passes (including the analytically derived ones from Exercises 2 and 3) into the full training loop, replacing `loss.backward()` entirely.

5.3 Exercise 1: Atomic Backpropagation Through the Graph

We begin by manually calculating the gradients for each intermediate tensor, starting from the loss and working backward to the inputs and parameters.

5.3.1 Backpropagating Through Loss Calculation (`logprobs` to `loss`)

The loss is defined as the negative mean of selected log probabilities:

```
loss = -logprobs[range(N), YB].mean()
```

Listing 48: Loss Calculation

Here, `N` is the batch size, and `YB` contains the correct indices for each example in the batch.

Derivation of `dlogprobs`: If `loss = -(A + B + C) / 3` for three numbers, then $\frac{dL}{dA} = -\frac{1}{3}$. Generalizing for `N` numbers (our batch size, e.g., 32), the derivative of the loss with respect to each *participating* `logprob` element is $-\frac{1}{N}$. For all other `logprob` elements that do not participate in the loss calculation (because they were not indexed by `YB`), their gradient with respect to the loss is intuitively zero, as changing them would not change the loss.

```
# dlogprobs (32, 27)
dlogprobs = torch.zeros_like(logprobs)
dlogprobs[range(N), YB] = -1.0 / N
cmp('logprobs', dlogprobs, logprobs)
```

Listing 49: Calculating dlogprobs

5.3.2 Backpropagating Through Log (`probs` to `logprobs`)

The `logprobs` are calculated by taking the element-wise logarithm of `probs`:

```
logprobs = torch.log(probs)
```

Listing 50: Logprobs Calculation

Derivation of `dprobs`: The local derivative of $\log(x)$ with respect to x is $\frac{1}{x}$. Applying the chain rule, we multiply the local derivative by the incoming gradient (`dlogprobs`). Intuitively, this step boosts the gradients for examples where the assigned probability of the correct character (`probs`) was very low. If `probs` is close to zero, $\frac{1}{\text{probs}}$ becomes very large, amplifying the gradient signal.

```
# dprobs (32, 27)
dprobs = (1.0 / probs) * dlogprobs
cmp('probs', dprobs, probs)
```

Listing 51: Calculating dprobs

5.3.3 Backpropagating Through Normalization (`counts`, `count_sum_inv` to `probs`)

The `probs` are obtained by an element-wise multiplication of `counts` and `count_sum_inv`:

```
probs = counts * count_sum_inv
```

Listing 52: Probabilities Calculation

Here, `counts` has shape (32, 27) and `count_sum_inv` has shape (32, 1). This implies an implicit broadcasting: `count_sum_inv` (a column tensor) is replicated 27 times horizontally to align with `counts` for element-wise multiplication.

Derivation of `dcount_sum_inv`: If $C = A \times B$ (scalar multiplication), then $\frac{dC}{dB} = A$. So, $\frac{d(\text{probs})}{d(\text{count\_sum\_inv})} = \text{counts}$ (local derivative for element-wise multiplication). Applying the chain rule: `dcount_sum_inv = counts * dprobs`. However, because `count_sum_inv` was replicated horizontally in the forward pass, its gradient must be the sum of all the gradients it contributed to. This means summing horizontally across dimension 1, while retaining the dimension to maintain the (32, 1) shape.

```
# dcount_sum_inv (32, 1)
dcount_sum_inv = (counts * dprobs).sum(dim=1, keepdim=
    True)
cmp('count_sum_inv', dcount_sum_inv, count_sum_inv)
```

Listing 53: Calculating dcount_sum_inv

Derivation of `dcounts` (first branch): Similarly, for $\frac{d(\text{probs})}{d(\text{counts})} = \text{count\_sum\_inv}$ (local derivative). Applying the chain rule: `dcounts = count_sum_inv * dprobs`. No additional summation is required here as `count_sum_inv` broadcasts correctly.

```
# dcounts (32, 27)
dcounts = count_sum_inv * dprobs
# Note: This is only a partial derivative, dcounts
    will be updated later
```

Listing 54: Calculating dcounts (partial)

5.3.4 Backpropagating Through Linear Layer 2 (`H`, `W2`, `B2` to `logits`)

The `logits` are the result of a matrix multiplication of `H` and `W2`, followed by an addition of `B2` (bias):

```
logits = H @ W2 + B2
```

Listing 55: Linear Layer 2 Calculation

Shapes: `H` (32, 64), `W2` (64, 27), `B2` (27,). `H @ W2` results in (32, 27). `B2` broadcasts from (27,) to (1, 27) then vertically to (32, 27).

Backpropagation Rules for Matrix Multiplication and Bias: Deriving these rules from first principles by writing out a small example $D = A@B + C$ and applying scalar chain rule for each element reveals a pattern:

- **Derivative with respect to `A` (`dH`):** $\frac{dL}{dA} = \frac{dL}{dD} @ B^T$.
  - In our case: `dH = dlogits @ W2.T`.
  - Dimension matching: `dlogits` (32, 27) @ `W2.T` (27, 64) $\rightarrow$ (32, 64), which matches `H`'s shape.
- **Derivative with respect to `B` (`dW2`):** $\frac{dL}{dB} = A^T @ \frac{dL}{dD}$.
  - In our case: `dW2 = H.T @ dlogits`.
  - Dimension matching: `H.T` (64, 32) @ `dlogits` (32, 27) $\rightarrow$ (64, 27), which matches `W2`'s shape.
- **Derivative with respect to `C` (`dB2`):** $\frac{dL}{dC}$ (for broadcasted bias) is the sum of $\frac{dL}{dD}$ across the broadcasted dimension.
  - In our case: `dB2 = dlogits.sum(dim=0)`.
  - Dimension matching: `dlogits` (32, 27) summed over `dim=0` (examples) $\rightarrow$ (27,), which matches `B2`'s shape.

```
# dH (32, 64), dW2 (64, 27), dB2 (27,)
dH = dlogits @ W2.T
dW2 = H.T @ dlogits
dB2 = dlogits.sum(dim=0)
cmp('H', dH, H)
cmp('W2', dW2, W2)
cmp('B2', dB2, B2)
```

Listing 56: Calculating dH, dW2, dB2

5.4 Exercise 2: Analytical Cross-Entropy Backward Pass

While the atomic backpropagation is good for understanding, it's often inefficient. For certain common operations, like cross-entropy loss, an analytical derivation of the gradient can result in a much simpler and faster backward pass.

5.4.1 Mathematical Derivation of `dlogits` for Cross-Entropy

The cross-entropy loss (for a single example) is given by:

$$L = -\log P_Y$$

where P_Y is the probability of the correct label Y, computed via softmax on the logits:

$$P_i = \frac{e^{L_i}}{\sum_j e^{L_j}}$$

We are interested in $\frac{\partial L}{\partial L_i}$, the derivative of the loss with respect to each logit L_i. The derivation distinguishes two cases:

- If $i = Y$ (the correct label): $\frac{\partial L}{\partial L_Y} = P_Y - 1$
- If $i \neq Y$ (any other label): $\frac{\partial L}{\partial L_i} = P_i$

This can be concisely stated as $\frac{\partial L}{\partial L_i} = P_i - \mathbb{I}(i = Y)$, where $\mathbb{I}$ is the indicator function. For a batch, the total loss is the mean of individual losses, so the gradients must also be scaled by $\frac{1}{N}$.

```
# dlogits (32, 27)
dlogits = F.softmax(logits, dim=1)  # Calculate P (
    probabilities)
dlogits[range(N), YB] -= 1          # Subtract 1 at
    the correct label positions
dlogits /= N                        # Scale by 1/N for
     the mean loss
cmp('logits', dlogits, logits)
```

Listing 57: Analytical dlogits for Cross-Entropy

5.4.2 Intuition of `dlogits` in Cross-Entropy

The `dlogits` gradient carries a profound intuitive meaning. If we visualize `dlogits`, we see that:

- For incorrect character positions, the gradient values are proportional to their predicted probabilities (P_i). These are "pulling down" forces.
- For the correct character position (Y), the gradient is $P_Y - 1$. Since P_Y is usually less than 1 (unless perfect prediction), this is a negative value, representing a "pulling up" force.
- The sum of gradients across each row (for each example) is zero (`sum(dlogits)` is 0). This signifies that the "push" on incorrect probabilities is perfectly balanced by the "pull" on the correct probability.
- The magnitude of the gradient (the push/pull force) is proportional to how much the network *mispredicted.* If a character was confidently mis-predicted, the corresponding negative gradient on that incorrect logit would be large, and the positive gradient on the correct logit would be equally large to compensate.

This dynamic push-pull mechanism, driven by cross-entropy, is what efficiently guides the neural network's training.

5.5 Exercise 3: Analytical Batch Normalization Backward Pass

Similar to cross-entropy, batch normalization's backward pass can be significantly simplified through analytical derivation. This exercise focuses on deriving `dH_PBN` from `dH_preact`, ignoring the derivatives for `BN_gain` and `BN_bias` for simplicity, as they are straightforward.

5.5.1 Mathematical Derivation of `dH_PBN`

The forward pass of batch normalization (simplified, ignoring γ and β) can be expressed as:

$$\mu = \frac{1}{N}\sum_j X_j \tag{1}$$

$$\sigma^2 = \frac{1}{N-1}\sum_j (X_j - \mu)^2 \tag{2}$$

$$\hat{X}_i = \frac{X_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \tag{3}$$

$$Y_i = \gamma \hat{X}_i + \beta \quad \text{(We are ignoring } \gamma \text{ and } \beta \text{ in this derivation, so } Y_i \approx \hat{X}_i) \tag{4}$$

We have $\frac{\partial L}{\partial Y_i}$ (which is `dH_preact`) and want to find $\frac{\partial L}{\partial X_i}$ (which is `dH_PBN`). The computational graph shows that each X_i influences Y_i directly through $\hat{X}_i$, and indirectly through μ and σ^2.

The final analytical expression for $\frac{\partial L}{\partial X_i}$ (which maps to `dH_PBN` for each column in the batch) is complex but much more efficient than the atomic step-by-step approach.

```
# dH_PBN (32, 64)
# Complex analytical formula implementation
dBN_diff_sum = dBN_diff.sum(dim=0)
dBN_diff_mul_dBN_raw = (dBN_diff * BN_raw).sum(dim=0)
dBN_var_inv_cubed = (bn_var + 1e-5)**(-1.5)
dH_PBN = dBN_diff * BN_var_inv - (dBN_diff_sum *
    BN_var_inv +
        dBN_diff_mul_dBN_raw * dBN_var_inv_cubed *
            BN_diff) / N
cmp('hpreact', dH_PBN, h_preact)
```

Listing 58: Analytical dH_PBN for Batch Normalization

5.6 Exercise 4: Full Training Loop with Manual Backpropagation

The final exercise integrates all the manually derived backward passes (including the analytical ones for cross-entropy and batch normalization) into the full neural network training loop.

5.6.1 Putting It All Together

The process involves:

1. Re-initializing the neural network parameters from scratch.
2. Replacing `loss.backward()` with the sequential calls to our manually implemented gradient calculations (`dlogprobs` all the way back to `dC`).
3. Disabling PyTorch's autograd for the parameter update step using `with torch.no_grad():` to improve efficiency, as we are managing gradients manually.

4. Updating the parameters using our manually computed `grad` values (e.g., `p.data += -learning_rate * grad`). The parameters and their respective gradients are zipped together for this update.
5. Calibrating the running mean and variance for batch normalization after the training loop.

5.6.2 Results and Significance

When the full training loop with manual backpropagation runs, it achieves a loss very similar to, if not identical to, what would be obtained using PyTorch's `loss.backward()`. The model also produces similarly decent samples.

The key takeaway is that the entire backward pass for this neural network, incorporating the simplified analytical expressions for cross-entropy and batch normalization, can be implemented in approximately 20 lines of code. This demonstrates complete visibility and control over the network's gradient computation, removing the "black box" nature of autograd.

```
# Full manual backward pass (conceptual structure)
# Replace loss.backward() with this block
dlogprobs = ...   # (Analytical from Exercise 2)
dprobs = ...
dcount_sum_inv = ...
dcount_sum = ...
dcounts = ...
dnorm_logits = ...
dlogits = ...     # (Analytical from Exercise 2)
dH, dW2, dB2 = ...
dH_preact = ...
dBN_gain, dBN_raw, dBN_bias = ...
dBN_diff, dBN_var_inv = ...
dBN_var = ...
dBN_diff_2 = ...
dBN_diff = ...
dH_PBN = ...      # (Analytical from Exercise 3)
dM_cat, dW1, dB1 = ...
dM = ...
dC = ...

# Store gradients in a list in parameter order
grads = [dC, dW1, dB1, dBN_gain, dBN_bias, dW2, dB2]

# Parameter update loop
with torch.no_grad():
    for p, grad in zip(parameters, grads):
        p.data += -learning_rate * grad
```

Listing 59: Manual Backward Pass (Simplified View of Exercise 4 Block)

5.7 Conclusion

This lecture provides a comprehensive understanding of how backward passes are implemented and how gradients flow through a neural network. By manually deriving and implementing the gradients for diverse layers, including matrix multiplications, element-wise operations, activations, and complex layers like cross-entropy and batch normalization, we gain invaluable intuition into the mechanics of neural network training.

While in practical deep learning applications, automatic differentiation frameworks like PyTorch's `autograd` are the go-to tools for their efficiency and convenience, understanding the underlying manual backward pass is crucial for:

- **Effective Debugging**: When models don't learn as expected, this knowledge is indispensable for pinpointing issues related to gradient flow (e.g., vanishing/exploding gradients, dead neurons).
- **Custom Layers and Operations**: If you need to implement custom layers or operations that `autograd` doesn't natively support, you'll have the skills to write their backward passes.
- **Deeper Understanding**: It transforms backpropagation from a "magical black box" into a transparent, understandable process, fostering a more robust foundation in deep learning.

This journey from `loss.backward()` to a fully manual, yet efficient, backpropagation implementation is a significant step towards becoming a "backprop ninja". The next lecture will delve into more complex architectures, such as Recurrent Neural Networks (RNNs) and LSTMs.

6 Lecture 6: Building makemore Part 5 - Building a WaveNet

Abstract

In this lecture, we continue our journey with `makemore`, our character-level language model. Our goal is to evolve from a simple Multi-Layer Perceptron (MLP) to a more complex, deeper architecture, specifically one that resembles the structure of a WaveNet. We address the limitations of our current approach by increasing context length and implementing progressive information fusion through a hierarchical, tree-like architecture. This approach mirrors the WaveNet architecture published by DeepMind in 2016, demonstrating how deeper models can gradually incorporate context rather than squashing all input information into a single hidden layer immediately.

6.1 Introduction: Towards Deeper, Hierarchical Models

Our current `makemore` model is a multi-layer perceptron that takes three previous characters as input and attempts to predict the fourth. This architecture uses a single hidden layer, but we want to address two key limitations:

1. **Increased Context**: We want to take more than just three characters as input.

2. **Progressive Information Fusion**: Instead of squashing all input information into a single hidden layer immediately, we desire a deeper model that fuses information progressively.

By implementing these changes, we will arrive at an architecture very similar to a WaveNet. WaveNet, published by DeepMind in 2016, is fundamentally an auto-regressive language model, but it was designed to predict audio sequences rather than character or word sequences. However, its core modeling setup, predicting the next element in a sequence, is identical to our task. The WaveNet architecture uses an interesting hierarchical, tree-like approach to prediction, which is what we will implement.

6.2 Recap of Previous `makemore` Implementation

Our starting code for Part 5 is based on Part 3, as Part 4 was a separate exercise focused on manual backpropagation.

6.2.1 Data Generation and Processing

We read a dataset of words and process them into individual examples. The data generation code remains unchanged. We currently have 182,000 examples, where each example consists of three characters used to predict the fourth.

6.2.2 Module-Based Architecture

In Part 3, we began developing our code around modular `Layer` classes, such as `Linear`, `BatchNorm1d`, and `Tanh`. This approach treats layers as "building blocks," similar to Lego bricks, allowing us to stack them to form neural networks and feed data between them. These custom layers were designed to mimic the API signatures of PyTorch's `torch.nn` modules.

- `Linear` **Layer**: Performs a matrix multiplication in its forward pass.
- `BatchNorm1d` **Layer**: This layer is notable for several reasons:
  - It maintains `running_mean` and `running_variance` statistics, which are updated via exponential moving average during the forward pass and are not trained via backpropagation.
  - It has different behaviors during `train` time and `eval` (evaluation) time, requiring careful state management. Forgetting to set the correct mode is a common source of bugs.
  - It couples the computation across examples within a batch, which is unusual as batches are typically seen only for efficiency. This coupling is for controlling activation statistics.
  - Its stateful nature (due to running statistics and train/eval modes) can lead to bugs, especially if means and variances haven't settled.
- `Tanh` **Layer**: A simple element-wise activation function.

The global random number generator (RNG) object, previously passed into layers, has been removed for simplicity, using a single global `torch.Generator` initialized once.

Our current neural network structure is:

- An embedding table `C`.
- A sequence of layers: `Linear` → `BatchNorm1d` → `Tanh` → `Linear` (output layer).
- The output layer weights are scaled down to prevent large errors at initialization.

The model has about 12,000 parameters. The optimization process (Adam optimizer, learning rate decay) remains identical to previous parts.

6.2.3 Current Performance Baseline

Before evaluating, all layers (specifically `BatchNorm1d`) must be set to `training=False`. Our current validation loss is around 2.10, which is "fairly good" but can be improved. Sampled names like `Yvon`, `kilo`, `Pros`, `Alaia` are "not unreasonable" but "not amazing".

6.3 Code Refinements and "PyTorch-ification"

We will now refactor our code to make it more organized, robust, and closer to PyTorch's standard practices.

6.3.1 Fixing the Loss Plot

The loss plot can be very "jagged" or "thick" due to small batch sizes (e.g., 32 batch elements) leading to high variance in individual batch losses. To get a more representative view of the loss curve, we can average consecutive loss values.

```
# Example of smoothing a loss list
loss_list = [0.1, 0.2, ..., 0.9, 0.8, 0.7, ...]  # Assume a long list of losses

# Convert to tensor and view as rows of 1000 consecutive elements
# If loss_list has 200,000 elements, this becomes a 200x1000 tensor
smoothed_losses = torch.tensor(loss_list).view(-1, 1000).mean(1)

# Now, plt.plot(smoothed_losses) will produce a much smoother graph
```

Listing 60: Example of smoothing a loss list

This technique is very helpful as it averages consecutive values into rows, then takes the mean along those rows. The resulting plot shows learning rate decay reducing energy and helping the model settle into a local minimum.

6.3.2 Encapsulating Embedding and Flattening as Modules

Previously, the embedding table lookup and the `view` (flattening/concatenation) operation were special-cased outside our list of `layers`. To further "PyTorch-ify" our code, we create dedicated `Embedding` and `Flatten` modules.

- `Embedding` **Module**: Handles the lookup of character embeddings. This mirrors `torch.nn.Embedding`.
- `Flatten` **Module**: Rearranges character embeddings into a single long row, effectively concatenating them. This `view` operation is computationally free in PyTorch as it doesn't copy memory. This mirrors `torch.nn.Flatten`.

```
# Custom Embedding Module
class Embedding:
    def __init__(self, num_embeddings, embedding_dim):
        self.weight = torch.randn((num_embeddings,
            embedding_dim), generator=g)

    def __call__(self, IX):
        return self.weight[IX]

    def parameters(self):
        return [self.weight]

# Custom Flatten Module
class Flatten:
    def __call__(self, x):
        # x is B x T x C (Batch, Time, Channels)
        # We want to flatten T and C into a single
            dimension
        B, T, C = x.shape
        return x.view(B, T * C)  # Or x.view(B, -1)

    def parameters(self):
        return []  # No trainable parameters
```

Listing 61: Custom Embedding Module

By incorporating these, the embedding table `C` becomes `self.weight` inside an `Embedding` module, and the forward pass is substantially simplified as these operations are now handled within our `layers` list.

6.3.3 Introducing the `Sequential` Container

PyTorch's `torch.nn` library includes "containers" like `Sequential`, which organize layers into lists and pass input sequentially through them. We implement our own `Sequential` module to manage our list of layers.

```
# Custom Sequential Module
class Sequential:
    def __init__(self, layers):
        self.layers = layers

    def __call__(self, x):
        for layer in self.layers:
            x = layer(x)
        return x

    def parameters(self):
        # Collect parameters from all child modules
        params = []
        for layer in self.layers:
            params.extend(layer.parameters())
        return params
```

Listing 62: Custom Sequential Module

This allows us to replace the "naked list of layers" with a `model` object, which is an instance of `Sequential`. This simplifies parameter collection (`model.parameters()`) and the forward pass (`model(xB)`).

6.3.4 Debugging the `BatchNorm1d` Sampling Issue

After refactoring, a bug emerged during sampling. When `BatchNorm1d` is in `training` mode and receives a batch with a single example, it attempts to estimate variance from a single number, which results in `NaN` (Not a Number). The variance of a single number is undefined. This `NaN` then pollutes subsequent calculations.

The fix is to ensure that all layers, especially `BatchNorm1d`, are correctly set to `eval` mode (by calling `model.eval()`) when evaluating or sampling from the model. If `BatchNorm1d` is in `eval` mode, it uses its stored `running_mean` and `running_variance` instead of calculating batch statistics, thus avoiding the `NaN` issue with single-example batches.

6.4 Implementing the WaveNet-like Hierarchical Architecture

Our current MLP crushes all input characters into a single layer too quickly. The WaveNet architecture offers a solution by progressively fusing information from previous characters in a tree-like, hierarchical manner.

6.4.1 Motivation for Hierarchical Fusion

Instead of immediately squashing all 8 characters (our new `block_size`) into a single wide hidden layer, we want the network to slowly fuse information. For instance, two characters are fused into a bigram representation, then two bigrams into a four-character chunk, and so on. This allows the network to get deeper while gradually incorporating context.

6.4.2 Increasing `block_size`

First, we adjust the `block_size` (the number of context characters) from 3 to 8. This means our model now takes 8 characters to predict the 9th. Even with the simple MLP structure, increasing context length from 3 to 8 characters already improved validation loss from 2.10 to 2.02. This serves as our new rough baseline performance.

6.4.3 Revisiting `Linear` and `Flatten` for Hierarchical Processing

Our current `Linear` layer expects a 2D input (e.g., `batch_size x total_features`), where `total_features` is `block_size * embedding_dim`. However, PyTorch's matrix multiplication (used in `Linear`) is more flexible: it operates only on the *last* dimension of the input tensor, treating all preceding dimensions as "batch" dimensions. This means we can have multiple "batch" dimensions and perform parallel matrix multiplications over them.

This property is key for hierarchical fusion. We don't want to flatten all 8 characters into an 80-dimensional vector (`8 * 10`) and multiply it immediately. Instead, we want to group them (e.g., two characters at a time), process these groups in parallel, and then fuse the results.

The Need for `FlattenConsecutive` Currently, our `Flatten` module takes a `B x T x C` tensor (e.g., `4 x 8 x 10`) and flattens it into `B x (T*C)` (e.g., `4 x 80`). For hierarchical processing, we want `Flatten` to output `B x (T/N) x (N*C)`, where `N` is the number of consecutive elements we want to fuse (e.g., `N=2` for fusing pairs).

For example, if `B=4`, `T=8`, `C=10`, and we want to fuse `N=2` consecutive character embeddings, the input to `FlattenConsecutive` is `4 x 8 x 10`. We want the output to be `4 x 4 x 20` (4 examples, 4 groups of 2 characters, each group becoming a 20-dimensional vector because $2 \times 10 = 20$).

```python
# Custom FlattenConsecutive Module
class FlattenConsecutive:
    def __init__(self, n):
        # n: number of consecutive elements to
            concatenate in the last dimension
        self.n = n

    def __call__(self, x):
        # x shape: B x T x C (Batch, Time/
            SequenceLength, Channels/EmbeddingDim)
        B, T, C = x.shape

        # Desired output shape: B x (T // n) x (C * n)
        # (T // n) creates the new 'group' dimension (
            e.g., 8 chars / 2 = 4 groups)
        # (C * n) combines the n consecutive elements
            into a single vector
        x = x.view(B, T // self.n, C * self.n)

        # Handle spurious dimension of 1, e.g., if T /
            / n results in 1
        # This occurs if the entire sequence length 'T
            ' is handled as one group
        # (e.g., block_size = 8, n = 8, then T // n =
            1)
        # In such cases, we want to return a 2D tensor (B x (C*n))
        # rather than a 3D tensor with a middle dimension of 1 (B x 1 x (C*n))
        if x.shape[1] == 1:
            x = x.squeeze(1)  # Squeeze out the middle dimension if it's 1

        return x

    def parameters(self):
        return []  # No trainable parameters
```

Listing 63: Custom FlattenConsecutive Module

Building the Hierarchical Model Structure With `FlattenConsecutive`, we can now construct a multi-layer hierarchical model. Each layer will:

1. Apply `FlattenConsecutive(2)` to fuse pairs of elements from the previous layer's output.
2. Pass through a `Linear` layer (whose input size is now `N * C_prev`).
3. Apply `BatchNorm1d`.
4. Apply `Tanh`.

The model will now look like this: `Embedding` → `FlattenConsecutive(2)` → `Linear` → `BatchNorm1d` → `Tanh` → `FlattenConsecutive(2)` → `Linear` → `BatchNorm1d` → `Tanh` → `FlattenConsecutive(2)` → `Linear` → `BatchNorm1d` → `Tanh` → `Linear` (output).

For example, using `block_size = 8`, `n_embed = 10`, and `n_hidden = 68`:

- Initial `xB` is `B x 8` (batch of 8 characters).
- `Embedding` outputs `B x 8 x 10`.
- `FlattenConsecutive(2)` outputs `B x 4 x 20` (4 groups of 2 characters, each fused into 20 dimensions).
- `Linear` (input 20, output 68) transforms to `B x 4 x 68`.
- `BatchNorm1d` and `Tanh` preserve `B x 4 x 68`.
- Next `FlattenConsecutive(2)` outputs `B x 2 x (2*68) = B x 2 x 136`.
- Next `Linear` (input 136, output 68) transforms to `B x 2 x 68`.
- `BatchNorm1d` and `Tanh` preserve `B x 2 x 68`.
- Next `FlattenConsecutive(2)` outputs `B x 1 x (2*68) = B x 1 x 136`. This dimension of 1 will be squeezed out to `B x 136`.
- Next `Linear` (input 136, output 68) transforms to `B x 68`.
- `BatchNorm1d` and `Tanh` preserve `B x 68`.
- Final `Linear` (input 68, output `vocab_size`) transforms to `B x vocab_size`. This is the final output logits.

This example effectively creates a three-layer hierarchical neural network. With `n_hidden = 68`, the total parameter count is about 22,000, similar to our initial MLP, allowing for a fair architectural comparison.

6.5 Correcting `BatchNorm1d` for Higher-Dimensional Inputs

The current `BatchNorm1d` implementation was designed assuming 2D inputs of shape `N x D` (Batch size x Features). However, with the hierarchical architecture, `BatchNorm1d` now receives 3D inputs, such as `32 x 4 x 68` (Batch x Groups x Features).

6.5.1 The Bug and Its Consequences

Our `BatchNorm1d` calculates mean and variance by reducing only over the *zeroth* dimension (the batch dimension). For a 3D input `32 x 4 x 68`, this means:

- `mean` and `variance` will have shapes like `1 x 4 x 68`.
- The `running_mean` and `running_variance` will also be `1 x 4 x 68`.

This implies that `BatchNorm1d` is maintaining separate statistics for each of the 4 "group" positions independently, rather than a single set of statistics for each of the 68 channels across all batch *and* group dimensions. In essence, it's normalizing `4 * 68` channels independently, with each estimate using only 32 numbers, instead of 68 channels, where each estimate uses `32 * 4` numbers. This makes the estimates less stable and "wiggly".

6.5.2 The Fix

`torch.mean` and `torch.var` (used internally by `BatchNorm1d`) can take a tuple of dimensions to reduce over. To correctly normalize across both the batch dimension (0) and the groups dimension (1), we need to compute the mean and variance over `(0, 1)`.

The output of such a reduction (e.g., `x.mean((0, 1))`) will be `1 x 1 x 68` for a 3D input of `B x T' x C'`. This correctly maintains 68 means/variances, one for each channel, estimated from all `B * T'` elements.

```python
# Adaptation for BatchNorm1d
class BatchNorm1d:
    # ... (existing __init__ and parameters methods)
        ...

    def __call__(self, x):
        # x.shape: B x D or B x T' x D' (where D or D'
            are number of channels)

        # Determine dimensions to reduce over based on
            input dimensionality
        if x.ndim == 2:
            dim_to_reduce = 0
        elif x.ndim == 3:
            dim_to_reduce = (0, 1)  # Reduce over
                batch and second (group) dimension
        else:
            raise ValueError(f"Unsupported input
               dimensionality: {x.ndim}")

        # ... (rest of the BatchNorm1d logic,
           replacing 0 with dim_to_reduce) ...
        # Example for calculating batch_mean:
        # batch_mean = x.mean(dim=dim_to_reduce,
           keepdim=True)
        # batch_var = x.var(dim=dim_to_reduce, keepdim
           =True)
```

Listing 64: Adaptation for BatchNorm1d

6.5.3 Departure from PyTorch API

It's important to note that our `BatchNorm1d` implementation now slightly deviates from `torch.nn.BatchNorm1d`. PyTorch's `BatchNorm1d` expects 3D inputs in the format `N x C x L` (Batch x Channels x Sequence Length), meaning it assumes channels are in the *middle* dimension and reduces over dimensions `(0, 2)`. Our implementation assumes channels are in the *last* dimension (`N x L x C`) and reduces over `(0, 1)`. This deviation is acceptable for our purposes as it aligns with our `FlattenConsecutive` design.

6.5.4 Performance After Bug Fix

After fixing `BatchNorm1d`, we observed a slight improvement in validation loss, from 2.029 to 2.022. This improvement is attributed to the more stable estimates of mean and variance, as they are now calculated using `32 * 4` numbers (for each channel) instead of just 32.

6.6 Scaling Up and Future Directions

With the more robust and general architecture, we can push performance further by increasing the network's capacity. For example, by increasing `n_embed` from 10 to 24 and adjusting `n_hidden`, the model now has 76,000 parameters. This led to a validation loss of 1.993, crossing the 2.0 territory.

However, further improvements are hampered by longer training times and the lack of a proper "experimental harness" for systematic hyperparameter tuning.

6.6.1 Relation to Convolutional Neural Networks (CNNs)

The WaveNet paper uses "dilated causal convolution layers". What we have implemented is the *logical structure* of the WaveNet's hierarchical fusion. The use of convolutions is primarily for **efficiency**, not to change the fundamental computation of the model.

In our current implementation, if we want to predict the next character for all positions in a name (e.g., DeAndre, 8 examples), we would call our model 8 independent times in a Python loop. Convolutional layers, on the other hand, allow us to "slide" this model efficiently over the input sequence, performing this loop inside highly optimized CUDA kernels. Furthermore, convolutions enable **variable reuse**. For instance, a node's computed value might be the right child of one computation tree and the left child of another. In a naive loop, this value would be recomputed, but convolutions reuse it efficiently. So, what we've built is akin to a single "black structure" calculation, while convolutions allow us to calculate all orange outputs (all positions) simultaneously.

6.6.2 Key Takeaways and Development Process Notes

1. **Module-Based Development**: We've reinforced the concept of building neural networks using modular "Lego building blocks" (layers and containers), essentially re-implementing parts of `torch.nn`. This understanding prepares us to use `torch.nn` directly in future work.
2. **Deep Learning Development Workflow**:
   - **Documentation Challenges**: PyTorch's documentation can be inconsistent, incomplete, or even incorrect. Developers often have to do their best with what's available.
   - **Shape Gymnastics**: A significant amount of time is spent ensuring that tensor shapes align across layers (e.g., 2D, 3D, 4D, `NCL` vs `NLC` conventions). This is a common, often messy, part of deep learning development.
   - **Prototyping in Notebooks**: It's common practice to prototype layer implementations and debug shapes in Jupyter notebooks. Once satisfied, the code is transferred to a more structured repository (e.g., VS Code) for running full experiments.

6.6.3 Future Lectures and Challenges

This lecture unlocks several exciting future topics:

1. Implementing `Conv1d` layers for efficiency.
2. Exploring residual connections and skip connections, as seen in WaveNet and other architectures.
3. Setting up a robust experimental harness for systematic hyperparameter tuning, which is crucial for typical deep learning workflows.
4. Covering other important architectures like Recurrent Neural Networks (RNNs), LSTMs, GRUs, and of course, Transformers.

Finally, a challenge: try to beat the current validation loss of 1.993! There's likely still room for improvement by tuning hyperparameters, allocating channel budgets differently, or even implementing more features from the original WaveNet paper (like gated linear units). It's not obvious that this hierarchical architecture will necessarily outperform a highly-tuned simple MLP, so experimentation is key.

7 Lecture 7: Building GPT from Scratch

Abstract

This lecture provides a comprehensive guide to building a Generatively Pre-trained Transformer (GPT) from scratch. We explore the fundamental architecture behind Chat GPT and similar language models, implementing each component step-by-step. Starting with a simple bigram model as a baseline, we progressively build up to a full Transformer architecture, covering tokenization, self-attention mechanisms, multi-head attention, feed-forward networks, and the complete training pipeline. By the end, we'll have a working character-level language model trained on Shakespeare's works that demonstrates the core principles powering modern large language models.

7.1 Introduction to Chat GPT and Language Models

By now, you've likely encountered Chat GPT, a system that has profoundly impacted the AI community and the world by enabling interaction with AI through text-based tasks. For instance, you can prompt Chat GPT to generate a haiku about the importance of understanding AI or even a humorous breaking news article about a leaf falling from a tree. A crucial characteristic to note is its *probabilistic nature*: the system can produce slightly different, yet equally valid, outputs for the exact same prompt.

Fundamentally, Chat GPT is what we call a **language model**. It excels at modeling sequences—whether of words, characters, or more generally, tokens—and understanding how these elements typically follow each other within a language like English. From its perspective, its primary role is to *complete a sequence*: you provide the beginning, and it generates the continuation.

7.1.1 The Transformer Architecture: The Brain Behind GPT

The underlying neural network performing the heavy lifting for Chat GPT is the **Transformer architecture**. This groundbreaking architecture was first introduced in a seminal 2017 paper titled "Attention Is All You Need". The very acronym GPT—which stands for "Generatively Pre-trained Transformer"—directly acknowledges this core component, with "Transformer" referring to the neural network. Although the Transformer was initially proposed and designed within the context of machine translation, its profound impact was unanticipated by its authors. Within five years, this architecture, with only minor modifications, was adopted and copy-pasted into a vast array of AI applications, fundamentally reshaping the field. At its core, it is the fundamental neural network driving Chat GPT.

7.1.2 Our Goal: Building a Character-Level Language Model

While Chat GPT is a highly sophisticated, production-grade system trained on a substantial portion of the internet and refined through intricate pre-training and fine-tuning stages, our objective in this lecture is more educational. We aim to build something *like* Chat GPT, specifically a **Transformer-based language model** that operates at the **character level**. This scaled-down approach is incredibly insightful for understanding the internal workings of these powerful systems.

Instead of a massive internet dataset, we will work with a smaller, more manageable collection of text: **Tiny Shakespeare**. This dataset comprises a concatenation of all of Shakespeare's works, contained within a single 1MB file. Our task will be to train a Transformer to model how these characters follow each other, enabling it to produce new character sequences that resemble Shakespearean language. For example, given a sequence like "verily my lor", the Transformer will learn to predict characters like "d" as the next likely element.

7.2 Data Preparation and Tokenization

Our initial step in building our character-level language model involves preparing the Tiny Shakespeare dataset for the neural network.

7.2.1 Loading and Inspecting the Data

The `tiny_shakespeare.txt` file, which is about 1 megabyte in size, contains approximately 1 million characters. Before we dive into modeling, it's good practice to load and inspect the raw text.

```
import torch

# Download the tiny shakespeare dataset if not already present
# !wget https://raw.githubusercontent.com/karpathy/makemore/master/tinyshakespeare.txt
with open('tinyshakespeare.txt', 'r', encoding='utf-8') as f:
    text = f.read()

print(f"Length of dataset in characters: {len(text)}")
# Expected output: Length of dataset in characters: 1115394

print(text[:1000])  # Print the first 1000 characters to get a feel for the data
# Expected output will be the beginning of Shakespeare's text.
```

Listing 65: Loading and inspecting Tiny Shakespeare data

7.2.2 Vocabulary and Character-Level Tokenization

To process the text with a neural network, we need to convert it into a numerical representation. First, we identify all unique characters present in the dataset to form our **vocabulary**. For a character-level model, each of these unique characters will be treated as an individual element (or token) in our sequences.

```
chars = sorted(list(set(text)))  # Get unique characters and sort them
vocab_size = len(chars)          # Determine the size of our vocabulary
print(''.join(chars))
# Example output: !$&',-.3:;?ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

```python
print(f"Vocabulary size: {vocab_size}")
# Example output: Vocabulary size: 65 (indicating 65 unique characters)
```

Listing 66: Creating vocabulary and sorting characters

Tokenization Strategy **Tokenization** is the process of converting raw text (a string) into a sequence of integers based on a predefined vocabulary of possible elements. Different strategies exist:

- **Character-level tokenization**: This is the simplest approach, directly translating individual characters into integers. While straightforward and using a small vocabulary, it results in very long sequences of integers. This is the method we will adopt for our educational model.
- **Subword tokenization**: This is widely used in practice by models like GPT. Techniques like Google's SentencePiece or OpenAI's Byte Pair Encoding (BPE), implemented in their `tiktoken` library, tokenize text into subword units rather than whole words or individual characters. This strikes a balance, offering a larger vocabulary (e.g., 50,000 tokens for GPT-2) and shorter integer sequences compared to character-level models.
- **Word-level tokenization**: This approach encodes entire words into integers.

For our character-level language model, we'll implement a simple encoder-decoder pair:

```python
stoi = {ch:i for i,ch in enumerate(chars)}  # Map character to integer (string-to-integer)
itos = {i:ch for i,ch in enumerate(chars)}  # Map integer to character (integer-to-string)
encode = lambda s: [stoi[c] for c in s]     # Encoder function: takes a string, returns a list of integers
decode = lambda l: ''.join([itos[i] for i in l])  # Decoder function: takes a list of integers, returns a string

print(encode("hi there"))
# Example output: (These are the integer representations of 'h', 'i', ' ', 't', 'h', 'e', 'r', 'e')
print(decode(encode("hi there")))
# Example output: hi there (Demonstrates the reversible nature of encoding/decoding)
```

Listing 67: Character-level encoder and decoder

Encoding the Entire Dataset Once our encoder is defined, we apply it to the entire Tiny Shakespeare text to transform it into a single, massive sequence of integers, which is then wrapped in a PyTorch tensor.

```python
data = torch.tensor(encode(text), dtype=torch.long)  # Encode the entire text and convert to a PyTorch tensor
print(data.shape, data.dtype)
# Example output: torch.Size([1115394]) torch.int64 (A 1D tensor of 1.1 million 64-bit integers)
print(data[:1000])  # Display the first 1000 encoded characters
```

Listing 68: Encoding the full dataset into a PyTorch tensor

7.2.3 Train and Validation Split

A standard practice in machine learning is to separate the dataset into training and validation sets. The **training data** (90% in our case) is used to teach the model, while the **validation data** (the remaining 10%) is held back and used to assess the model's performance on unseen examples. This helps us understand the extent to which our model might be *overfitting* (memorizing the training data too well) versus truly learning generalizable patterns to produce "Shakespeare-like" text.

```python
n = int(0.9*len(data))  # Calculate the split point for 90% training data
train_data = data[:n]   # The first 90% of the data
val_data = data[n:]     # The last 10% of the data
```

Listing 69: Splitting data into train and validation sets

7.3 Batching and Context Length

When training a Transformer, it's computationally prohibitive and inefficient to feed the entire text dataset into the model at once. Instead, we work with smaller, manageable segments of the data, referred to as "chunks" or "blocks".

7.3.1 Block Size and Multiple Examples

The `block_size` (also known as `context_length`) dictates the maximum number of characters the Transformer will consider as context when making a prediction. Each sampled chunk of data, specifically `block_size + 1` characters long, effectively contains *multiple independent examples* packed within it. For instance, if `block_size` is 8, a 9-character chunk ($X = [c_1, c_2, \ldots, c_9]$) will yield 8 input-target pairs:

- Input: $[c_1]$, Target: c_2
- Input: $[c_1, c_2]$, Target: c_3
- ...
- Input: $[c_1, c_2, \ldots, c_8]$, Target: c_9

This strategy is not just for efficiency; it's crucial for training the Transformer to be adept at predicting the next character from contexts of varying lengths, from as little as one character up to the full `block_size`. This versatility is particularly useful during inference, where generation might begin with minimal context.

7.3.2 Batch Dimension for Parallel Processing

To maximize the utilization of modern parallel computing hardware, particularly GPUs, we process multiple `chunks` simultaneously. These multiple chunks are stacked together into a single tensor, forming a **batch**. Each chunk within a batch is processed completely independently, with no communication between them.

```python
# --- Hyperparameters for batching ---
batch_size = 4  # Number of independent sequences processed in parallel
block_size = 8  # Maximum context length for predictions

# --- Function to retrieve a batch ---
def get_batch(split):
    data = train_data if split == 'train' else val_data  # Select data based on split
    # Generate `batch_size` random starting indices within the selected data
    # ensuring there's enough room for a `block_size + 1` chunk
    ix = torch.randint(len(data) - block_size, (batch_size,))

    # Stack the input chunks (first `block_size` characters)
    x = torch.stack([data[i:i+block_size] for i in ix])
    # Stack the target chunks (characters offset by one from inputs)
    y = torch.stack([data[i+1:i+block_size+1] for i in ix])
    return x, y

# --- Example of retrieving and inspecting a batch ---
xb, yb = get_batch('train')
print('Inputs␣(xb):')
print(xb.shape)  # Expected output: torch.Size([4, 8]) (batch_size, block_size)
print(xb)
print('Targets␣(yb):')
print(yb.shape)  # Expected output: torch.Size([4, 8]) (batch_size, block_size)
print(yb)
```

Listing 70: Function to get a batch of data

The `xb` (inputs) tensor will have a shape of `(batch_size, block_size)`, and `yb` (targets) will have the same shape, serving as the desired next characters for each position in `xb`. A 4x8 batch, for instance, contains a total of 32 independent input-target examples for the model to learn from.

7.4 The Bigram Language Model (Baseline)

To begin our modeling journey, we'll implement the simplest possible neural network for language modeling: the **Bigram language model**. This model makes a prediction for the next character based solely on the identity of the single *current* character.

7.4.1 Model Architecture: `nn.Embedding`

The core component of our Bigram model is an embedding table, implemented using PyTorch's `nn.Embedding` module.

- `nn.Embedding(num_embeddings, embedding_dim)`: For a Bigram model, `num_embeddings` is set to `vocab_size` (our 65 unique characters), and `embedding_dim` is also set to `vocab_size`.
- When an integer representing a character (from our input `idx`) is passed to this embedding table, it acts as an index to "pluck out" a corresponding row (vector) from the table.
- The output, which we call `logits`, will have a shape of `(Batch, Time, Channels)`. In this context, `Channels` refers to the `vocab_size`, representing the scores for each possible next character in our vocabulary.

```python
import torch.nn as nn
from torch.nn import functional as F

class BigramLanguageModel(nn.Module):
    def __init__(self, vocab_size):
        super().__init__()
        # Each token directly reads off the logits for the next token from a lookup table.
        # This table is essentially a `vocab_size x vocab_size` matrix.
        self.token_embedding_table = nn.Embedding(vocab_size, vocab_size)

    def forward(self, idx, targets=None):
        # `idx` (inputs) and `targets` are both (B,T) tensors of integers.
        # `idx` represents the current batch of input sequences.
        logits = self.token_embedding_table(idx)  # Output shape: (B, T, C), where C is vocab_size

        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape
            # Reshape `logits` for `F.cross_entropy`. PyTorch's cross_entropy
            # expects predictions in a 2D format (N, C) or (B, C, T) for higher dimensions.
            # We flatten B*T positions into N, keeping C (channel/vocab_size) as the second dimension.
            logits = logits.view(B*T, C)
            # Flatten ‘targets‘ to match the ‘logits‘’s first dimension.
            targets = targets.view(B*T)
            # Calculate the cross-entropy loss between predicted logits and actual targets.
            loss = F.cross_entropy(logits, targets)

        return logits, loss

    def generate(self, idx, max_new_tokens):
        # ‘idx‘ is a (B, T) array of indices representing the current context (input sequence).
        for _ in range(max_new_tokens):
            # 1. Get predictions (logits) from the model based on the current context (‘idx‘).
            # We only need the forward pass here; loss is not computed as there are no ground truth targets.
            logits, loss = self(idx)

            # 2. Focus only on the last time step (‘-1‘ index in the ‘T‘ dimension).
            # These are the predictions for the very next token in each sequence in the batch.
            logits = logits[:, -1, :]  # Output shape: (B, C)

            # 3. Apply softmax to convert logits (raw scores) into probabilities.
            # ‘dim=-1‘ ensures softmax is applied across the vocabulary dimension for each batch element.
            probs = F.softmax(logits, dim=-1)  # Output shape: (B, C)

            # 4. Sample the next token from the probability distribution.
            # ‘torch.multinomial‘ draws ‘num_samples=1‘ from each row of ‘probs‘.
            idx_next = torch.multinomial(probs, num_samples=1)  # Output shape: (B, 1)

            # 5. Append the sampled index (‘idx_next‘) to the running sequence (‘idx‘).
            # ‘dim=1‘ concatenates along the time dimension, extending the context.
            idx = torch.cat((idx, idx_next), dim=1)  # Output shape: (B, T+1)
        return idx
```

Listing 71: BigramLanguageModel class

7.4.2 Loss Function and Initial State

To evaluate the quality of our model's predictions, we use the **Cross-Entropy Loss**. This is a common choice for classification problems and is equivalent to the negative log-likelihood loss. Intuitively, this loss function encourages the model to assign high scores (logits) to the correct next character.

For a completely random model with a vocabulary size of 65 characters, the expected negative log-likelihood loss (if it's predicting uniformly) would be $\ln(\text{vocab\_size})$, or $\ln(65) \approx 4.17$. An observed initial loss of approximately 4.87 for our untrained model indicates that its initial predictions are not perfectly uniform; there's a slight bias, but it's still largely guessing incorrectly, as expected for an uninitialized network.

7.4.3 Generation from the Model

The `generate` function facilitates the model's ability to produce new text sequences. It takes a starting sequence (e.g., a single token representing a newline character, `idx = torch.zeros((1, 1), dtype=torch.long)`) and iteratively extends it for a specified number of `max_new_tokens`.

When run with an untrained Bigram model, the generated text is expectedly "garbage" (random characters), as the model has not yet learned any patterns from the Shakespeare data.

7.5 The Core of Transformer: Self-Attention

The Bigram model's limitation lies in its inability for tokens to "talk to each other" and consider broader context. This is precisely the problem that **Self-Attention** is designed to solve. It enables tokens within a sequence to weigh the importance of other tokens when processing their own information, thereby building a richer, context-aware representation.

7.5.1 Mathematical Trick: Efficient Weighted Aggregation

Self-attention leverages a clever mathematical trick to efficiently perform weighted aggregations (or weighted sums/averages) of elements from the past within a sequence. The fundamental idea is that each token should gather information from all preceding tokens in its context, but crucially, it must *not* access information from future tokens in an auto-regressive (next-token prediction) scenario.

1. **Simple Averaging (Bag of Words)**: A naive starting point would be to simply average the feature vectors of all preceding tokens. This crude form of interaction is highly "lossy" regarding spatial arrangement but provides a basic form of communication.

2. **Matrix Multiplication for Weighted Sums**: The same weighted aggregation can be performed vastly more efficiently using matrix multiplication with a special lower-triangular matrix.

```python
# For a given sequence length T (e.g., 8, which is our block_size)
# Create a T x T matrix of ones
weights_matrix = torch.ones(T, T)
# Convert it into a lower-triangular matrix (elements above the main diagonal become zero)
weights_matrix = torch.tril(weights_matrix)
# Normalize each row so that its elements sum to 1. This makes the matrix multiplication
# perform an average of the corresponding rows from the input.
weights_matrix = weights_matrix / weights_matrix.sum(1, keepdim=True)

# When this ‘weights_matrix‘ (T, T) is matrix-multiplied with our input ‘x‘ (B, T, C)
# using batched matrix multiplication, PyTorch automatically handles the batch dimension.
# The operation becomes: ‘output = weights_matrix @ x‘
```

Listing 72: Efficient weighted aggregation using lower-triangular matrix

3. **Masked Softmax for Affinities**: The most sophisticated approach, used in self-attention, involves producing a “weights” matrix where elements reflect “affinities” or “interaction strengths”. This is achieved by:

   - Initializing a scores matrix (e.g., with zeros).
   - Applying a lower-triangular mask, setting elements corresponding to future tokens to negative infinity (`float('-inf')`). This ensures these positions receive zero weight after softmax, effectively preventing future communication.
   - Applying `softmax` along the last dimension (each row). `softmax(negative_infinity` evaluates to zero, so the masked elements become zero probabilities. The other elements become probabilities that sum to one, representing how much attention (weight) to give to each past token.

```python
# For a sequence length T (e.g., block_size)
raw_affinities = torch.zeros((T, T))  # Placeholder for initial interaction strengths
# Create the lower-triangular mask, which is ‘1‘ for current/past, ‘0‘ for future.
tril = torch.tril(torch.ones(T, T))
```

```
    # Apply the mask: set values where `tril` is 0 (i.e., future positions) to negative infinity.
    # This ensures they get 0 probability after softmax.
    masked_affinities = raw_affinities.masked_fill(tril == 0, float('-inf'))
    # Apply softmax along the last dimension to normalize affinities into probabilities (weights).
    attention_weights = F.softmax(masked_affinities, dim=-1)  # Each row sums to 1

    # This `attention_weights` matrix will then be used in a batched matrix multiplication
    # with the input `x` to perform a weighted aggregation: `output = attention_weights @ x`.
```

Listing 73: Masked softmax for attention weights

Crucially, in a full self-attention mechanism, these raw_affinities will not be constant zeros. Instead, they will be *data-dependent*, meaning their values will be dynamically computed based on how interesting tokens find each other.

7.5.2 Token and Positional Embeddings

For the Transformer, the input tokens undergo two types of embedding to create a rich representation:

1. **Token Embeddings**: Similar to our Bigram model, each character's identity is mapped to a vector. However, the embedding_dim is now a separate hyperparameter, n_embed (e.g., 32), which defines the dimensionality of the token's content representation.
2. **Positional Embeddings**: Attention mechanisms inherently process sets of vectors without any built-in sense of order or position. To inject this crucial spatial information, a separate nn.Embedding table is created for positions. This table maps each numerical position (from 0 up to block_size - 1) to a unique embedding vector.

These two embedding vectors (token and positional) are then simply *added* together for each token. This element-wise addition creates a combined representation, x, that contains both the semantic content of the token and its sequential position.

```
class LanguageModel(nn.Module):
    def __init__(self, vocab_size, n_embed, block_size):
        super().__init__()
        # Token embedding table: maps each character ID to an `n_embed`-dimensional vector
        self.token_embedding_table = nn.Embedding(vocab_size, n_embed)
        # Positional embedding table: maps each position ID (0 to block_size-1)
        # to an ‘n_embed‘-dimensional vector
        self.position_embedding_table = nn.Embedding(block_size, n_embed)
        # The final linear layer (language modeling head) projects ‘n_embed‘ back to ‘vocab_size‘ for logits
        self.lm_head = nn.Linear(n_embed, vocab_size)

    def forward(self, idx, targets=None):
        B, T = idx.shape  # Get batch size (B) and sequence length (T) from input ‘idx‘

        # 1. Generate token embeddings based on the character IDs in ‘idx‘
        token_embeddings = self.token_embedding_table(idx)  # Shape: (B, T, n_embed)

        # 2. Generate positional embeddings for the current sequence length ‘T‘
        # ‘torch.arange(T, device=device)‘ creates a sequence of integers [0, 1, ..., T-1]
        position_embeddings = self.position_embedding_table(torch.arange(T, device=device))  # Shape: (T, n_embed)

        # 3. Add token and positional embeddings. PyTorch's broadcasting ensures
        # the (T, n_embed) position_embeddings are added to each (T, n_embed) slice across the batch.
        x = token_embeddings + position_embeddings  # Shape: (B, T, n_embed)

        # In a complete Transformer, ‘x‘ would then be fed into self-attention and feed-forward blocks.
        # For this simplified illustration, it goes directly to the language modeling head.
        logits = self.lm_head(x)  # Output shape: (B, T, vocab_size)

        # (Loss calculation logic would be similar to the Bigram model, using ‘logits‘ and ‘targets‘)
        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape
            logits = logits.view(B*T, C)
            targets = targets.view(B*T)
            loss = F.cross_entropy(logits, targets)

        return logits, loss
```

Listing 74: Token and Positional Embeddings in the LanguageModel

7.5.3 Query, Key, Value (QKV) Mechanism in a Single Head

This is truly the "crux" of how self-attention works. For every single token (or "node") within a sequence, at each position, the model generates three distinct vectors through linear transformations applied to its combined token+positional embedding (x):

- **Query (Q)**: Conceptually, this vector represents "what am I looking for?". It's the "question" a token asks of all other tokens.
- **Key (K)**: This vector represents "what information do I contain?". It's the "answer" a token offers to other tokens' queries.
- **Value (V)**: This vector represents "what information should I communicate (or pass along) if another token finds my key interesting?". It's the actual content that gets aggregated.

The communication happens when **affinities** (or raw attention scores) between tokens are calculated. This is done by taking the **dot product of Queries and Keys**. If a query vector is "aligned" with a key vector (i.e., they are similar in direction and magnitude), their dot product will be high, indicating a strong affinity or mutual interest.

```
class Head(nn.Module):
    def __init__(self, head_size, n_embed, block_size, dropout_rate):
        super().__init__()
        # Linear layers to project input ‘x‘ into Key, Query, and Value vectors
        # Bias is typically set to False for these projections in Transformers
        self.key = nn.Linear(n_embed, head_size, bias=False)
        self.query = nn.Linear(n_embed, head_size, bias=False)
        self.value = nn.Linear(n_embed, head_size, bias=False)

        # ‘tril‘ (lower triangular mask) is a buffer, not a trainable parameter.
        # It's registered so it moves with the module to the correct device.
        self.register_buffer('tril', torch.tril(torch.ones(block_size, block_size)))

        self.head_size = head_size  # Store head_size for scaling
        self.dropout = nn.Dropout(dropout_rate)  # Dropout layer

    def forward(self, x):
```

```python
        B, T, C = x.shape  # Get batch size, sequence length, and embedding dimension
        k = self.key(x)    # (B, T, head_size) - Keys for all tokens in batch/sequence
        q = self.query(x)  # (B, T, head_size) - Queries for all tokens in batch/sequence

        # 1. Compute attention scores (affinities): Query @ Key^T
        # `(q @ k.transpose(-2, -1))` performs batched matrix multiplication.
        # Output shape: (B, T, T). Each (i,j) entry is affinity of query_i with key_j.
        # 2. Apply scaling: Divide by square root of head_size to control variance.
        att = (q @ k.transpose(-2, -1)) * (self.head_size**-0.5)

        # 3. Apply masking: Set affinities to future tokens to -infinity.
        # `self.tril[:T, :T]` ensures the mask adapts to the current sequence length `T`.
        att = att.masked_fill(self.tril[:T, :T] == 0, float('-inf'))

        # 4. Apply softmax: Normalize affinities into probability distribution (attention weights).
        att = F.softmax(att, dim=-1)  # (B, T, T) - rows sum to 1

        # 5. Apply dropout: Randomly drop some connections (regularization).
        att = self.dropout(att)

        v = self.value(x)  # (B, T, head_size) - Values for all tokens

        # 6. Perform weighted aggregation: Attention_weights @ Values
        # Output shape: (B, T, head_size) - contextualized representation for each token
        out = att @ v
        return out
```

Listing 75: Single Self-Attention Head implementation

7.6 Building the Full Transformer Block

A complete Transformer architecture is not just a single attention head. It intersperses powerful communication mechanisms (attention) with local, per-token computation (feed-forward networks). These components are then structured within "blocks" that can be stacked to form a deep network.

7.6.1 Multi-Head Attention

Instead of relying on a single attention head to capture all necessary relationships, **Multi-Head Attention** runs multiple attention “heads” in parallel. Each individual head computes its own set of Q, K, and V projections, calculates its own attention scores, and aggregates its own values.

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, num_heads, head_size, n_embed, block_size, dropout_rate):
        super().__init__()
        # Create a list of `Head` modules, each performing a single attention operation.
        self.heads = nn.ModuleList([Head(head_size, n_embed, block_size, dropout_rate) for _ in range(num_heads)])
        # A linear projection layer to combine the outputs of the multiple heads
        # It takes `num_heads * head_size` (which should equal `n_embed`) and projects it back to `n_embed`.
        self.proj = nn.Linear(num_heads * head_size, n_embed)
        self.dropout = nn.Dropout(dropout_rate)

    def forward(self, x):
        # 1. Run each attention head in parallel and collect their outputs.
        # Each h(x) produces (B, T, head_size).
        # 2. Concatenate the outputs of all heads along the last dimension (channel dimension).
        # Resulting shape: (B, T, num_heads * head_size), which is (B, T, n_embed).
        out = torch.cat([h(x) for h in self.heads], dim=-1)
        # 3. Apply the projection layer to the concatenated output.
        out = self.proj(out)
        # 4. Apply dropout for regularization.
        out = self.dropout(out)
        return out
```

Listing 76: MultiHeadAttention implementation

7.6.2 Feed-Forward Network (FFN)

After tokens have communicated and gathered contextual information through the attention mechanism, they need an opportunity to “think” or process that newly integrated information individually. This is the role of the **Feed-Forward Network** (also known as a Position-Wise Feed-Forward Network or a simple Multi-Layer Perceptron).

```python
class FeedFoward(nn.Module):
    def __init__(self, n_embed, dropout_rate):
        super().__init__()
        self.net = nn.Sequential(
            # First linear layer: expands
               dimensionality by a factor of 4 (as is
                common in Transformers)
            nn.Linear(n_embed, 4 * n_embed),
            nn.ReLU(),  # Non-linearity (GELU is also
               common, especially in OpenAI models)
            # Second linear layer: projects
               dimensionality back to `n_embed`
            nn.Linear(4 * n_embed, n_embed),
            nn.Dropout(dropout_rate),  # Dropout for
               regularization
        )

    def forward(self, x):
        # The network operates independently on each
           token's embedding (B, T, n_embed)
        return self.net(x)
```

Listing 77: FeedFoward implementation

7.6.3 Residual Connections and Layer Normalization

For training very deep neural networks, crucial architectural innovations are the use of **residual connections** (also known as **skip connections**) and **layer normalization**.

Residual connections establish a direct "superhighway" for gradients to flow unimpeded from the loss function all the way back to the input. The idea is that the main computational blocks (like Multi-Head Attention or the Feed-Forward Network) "fork off" from a main residual pathway, perform their transformation on the input `x`, and then their output is *added* back to the original `x` (`x = x + module(x)`).

Layer Normalization ensures that the inputs to subsequent layers maintain a stable distribution (e.g., zero mean and unit variance) during training, especially at initialization. A common and often more stable practice in modern Transformers is the "pre-norm" formulation, applying Layer Normalization *before* the subsequent transformation rather than after the residual connection.

A full Transformer block, integrating Multi-Head Attention, a Feed-Forward Network, Residual Connections, and Layer Normalization (using the pre-norm formulation), looks like this:

```
class Block(nn.Module):
    def __init__(self, n_embed, num_heads, block_size,
         dropout_rate):
        super().__init__()
        # Calculate head_size for each attention head
        head_size = n_embed // num_heads

        # Multi-Head Self-Attention layer
        self.sa = MultiHeadAttention(num_heads,
            head_size, n_embed, block_size,
            dropout_rate)
```

```python
        # Feed-Forward Network
        self.ffwd = FeedFoward(n_embed, dropout_rate)

        # Layer Normalization layers (pre-norm formulation)
        self.ln1 = nn.LayerNorm(n_embed)  # Applied before Multi-Head Attention
        self.ln2 = nn.LayerNorm(n_embed)  # Applied before Feed-Forward Network

    def forward(self, x):
        # Apply Layer Norm 1, then Multi-Head Self-Attention, then add residual connection
        x = x + self.sa(self.ln1(x))
        # Apply Layer Norm 2, then Feed-Forward Network, then add residual connection
        x = x + self.ffwd(self.ln2(x))
        return x
```

Listing 78: Transformer Block implementation (`Block` class)

7.7 Scaling Up the Model and Results

With all the core Transformer components in place (token/positional embeddings, scaled multi-head attention, feed-forward networks, residual connections, layer normalization, and dropout), we can now significantly scale up our model to observe its full potential on the Tiny Shakespeare dataset.

7.7.1 Hyperparameter Scaling

To push performance, we increase several hyperparameters, making the network considerably larger and more capable:

- `batch_size`: Increased from 4 to **64**.
- `block_size`: Increased from 8 to **256**, allowing the model to process and learn from much longer contexts.
- `n_embed`: The embedding dimension, increased from 32 to **384**. This makes the internal representations richer.
- `num_heads`: Increased to **6**. With `n_embed = 384`, each head will have a `head_size` of $384/6 = 64$ dimensions.
- `n_layer`: The number of Transformer `Block` layers, set to **6**. This creates a deeper network.
- `dropout`: Set to **0.2**, meaning 20% of intermediate calculations are randomly dropped during training.
- `learning_rate`: Slightly reduced because larger neural networks are often more sensitive to high learning rates.

7.7.2 Performance Improvement

After training with these scaled-up parameters (which might take about 15 minutes on a powerful A100 GPU, or longer on less capable hardware), the results are significantly improved. The validation loss drops to approximately **1.48**, a substantial decrease from the 2.07-2.08 range observed with earlier, smaller models.

The generated text becomes far more coherent and recognizable as Shakespearean, despite remaining semantically nonsensical. The model learns to mimic the style, sentence structure, and vocabulary patterns of Shakespeare's works with impressive fidelity. For instance, it might produce phrases like "is every crimp tap be a house" or "Oho sent me you mighty Lord," which superficially resemble English but lack logical meaning. This demonstrates the model's ability to "blabber on in Shakespeare-like manner".

7.8 GPT: A Decoder-Only Transformer

The Transformer model we have meticulously built is a **decoder-only Transformer**. This specific architecture means it fundamentally lacks two key components present in the original Transformer paper's full diagram: an "encoder" part and "cross-attention" blocks within its layers. Our model's blocks solely consist of self-attention and feed-forward networks.

The original "Attention Is All You Need" paper introduced the Transformer for *machine translation*. For this task, an **encoder-decoder architecture** is essential:

- **Encoder**: This part processes the input sequence (e.g., a French sentence). Crucially, the encoder blocks allow all tokens within the input sequence to communicate freely with each other (no triangular masking) to build a rich representation of the source sentence's content.
- **Decoder**: This part generates the output sequence (e.g., an English translation) in an auto-regressive fashion, meaning it predicts one token at a time. Like our model, it uses a triangular mask within its self-attention to prevent looking into the future. However, an encoder-decoder Transformer's decoder blocks also contain **cross-attention** layers. These cross-attention layers allow the decoder to "condition" its generation on the rich, encoded context provided by the encoder's output.

Our model, like OpenAI's GPT series, is a **decoder-only** Transformer because it is designed for **unconditioned text generation** (language modeling). It doesn't receive an external prompt to translate or condition upon; it simply learns to generate text that imitates the patterns of its training data. The presence of the triangular mask in its attention mechanisms makes it a "decoder" because it maintains the auto-regressive property, allowing it to predict subsequent tokens sequentially.

7.9 Conclusion

In this comprehensive lecture, we have meticulously explored and implemented a **decoder-only Transformer** from scratch, mirroring the fundamental architecture of Generative Pre-trained Transformer (GPT) models as laid out in the seminal 2017 "Attention Is All

You Need" paper. We successfully trained this model on the modest Tiny Shakespeare dataset, achieving sensible and increasingly coherent results that mimic the style of the input text.

Through this hands-on construction, we gained deep insights into the core components of the Transformer:

- The critical role of **token and positional embeddings** in providing both content and sequential information to the model.
- The intricate mechanics of **self-attention**, including the Query-Key-Value (QKV) mechanism, the importance of **scaled attention** for stable training, and the application of **causal masking** for auto-regressive generation.
- The power of **multi-head attention** in enabling diverse communication channels between tokens.
- The function of **feed-forward networks** for per-token computation.
- The crucial role of **residual connections** and **layer normalization** in ensuring the optimizability and stability of deep neural networks.
- The utility of **dropout** as a regularization technique to prevent overfitting.

All the core training code we developed is remarkably compact, typically around 200 lines. While our model is a "baby GPT" compared to the colossal scale of models like GPT-3 (which are 10,000 to a million times larger depending on how you measure parameters and data), the underlying architectural principles we've implemented are nearly identical.

It's important to reiterate that this lecture primarily focused on the **pre-training stage** of large language models. We did not delve into the subsequent, equally complex **fine-tuning stages** that are essential for aligning these models to specific tasks, such as generating answers to questions, detecting sentiment, or engaging in coherent dialogue (as seen in Chat GPT). These advanced stages often involve sophisticated techniques like supervised fine-tuning, reward model training, and reinforcement learning from human feedback (RLHF).

8 Lecture 8: The State of GPT - From Training to Effective Application

Abstract

This lecture outlines the current recipe for training GPT assistants and explores effective methods for their application. The rapidly growing ecosystem of Large Language Models (LLMs) and GPT assistants represents a significant area of development in Artificial Intelligence. We examine the multi-stage training pipeline from pre-training to reinforcement learning from human feedback (RLHF), and provide practical guidance on applying these models effectively through prompt engineering, tool integration, and fine-tuning strategies. It is important to note that this field is still very new and continues to evolve rapidly.

8.1 Introduction to GPT Assistants and Large Language Models

The training of GPT assistants typically follows a multi-stage recipe, which is currently an emerging standard. This process is largely sequential, with each stage building upon the previous one. Each stage in the training pipeline is powered by a specific dataset, driven by an algorithm (objective function), and results in a refined model.

8.1.1 The Four Major Training Stages

1. **Pre-training**: Focuses on creating a "base model".
2. **Supervised Fine-tuning (SFT)**: Develops an "SFT model".
3. **Reward Modeling**: Creates a "reward model".
4. **Reinforcement Learning (RL)**: Produces the final "assistant model".

It is crucial to understand that the pre-training stage is distinct and vastly more computationally intensive than the subsequent fine-tuning stages.

8.2 Stage 1: Pre-training (Achieving a Base Model)

This stage is the foundation and consumes approximately 99% of the total computational work, including compute time and FLOPs. It involves internet-scale datasets, thousands of GPUs, and can take months of training. The other three fine-tuning stages are comparatively smaller, requiring fewer GPUs and only hours or days.

8.2.1 Data Gathering (Data Mixture)

A massive amount of data is collected and mixed from various sources. Examples of datasets include:

- **Common Crawl**: A web scrape.
- **C4**: Also derived from Common Crawl.
- **High-Quality Datasets**: GitHub, Wikipedia, books, Archive, Stack Exchange, and similar sources.

These datasets are mixed and sampled according to specific proportions to form the training set for the GPT's neural network.

8.2.2 Tokenization

Before training, raw text from the internet must be pre-processed into sequences of integers. This process, called tokenization, is lossless and translates text into the native representation for GPTs. A common algorithm for this is Byte Pair Encoding (BPE), which iteratively merges text chunks into tokens.

8.2.3 Hyperparameters in Pre-training

Key hyperparameters govern the scale and capacity of the pre-trained models:

- **Vocabulary Size**: Typically around tens of thousands of tokens (e.g., 50,257 tokens mentioned).
- **Context Length**: The maximum number of integers (tokens) the GPT will consider when predicting the next integer in a sequence. This can range from 2,000 to 4,000, and even up to 100,000 tokens in modern models.
- **Number of Parameters**: While GPT-3 had 175 billion parameters, a model like LLaMA with 65 billion parameters can be significantly more powerful due to being trained on a much larger corpus of tokens (e.g., 1.4 trillion tokens for LLaMA vs. 300 billion for GPT-3). This indicates that the quantity of training data can outweigh raw parameter count in determining model power.
- **Transformer Network Architecture**: Parameters like the number of attention heads, dimension size, and number of layers define the Transformer neural network.
- **Training Resources**: Training a 65 billion parameter model (like LLaMA) can involve approximately 2,000 GPUs, 21 days of training, and cost several million dollars.

8.2.4 The Pre-training Process (Language Modeling Objective)

During pre-training, tokens are arranged into data batches, typically as two-dimensional arrays of size $B \times T$, where B is the batch size and T is the maximum context length. Documents are packed into rows and delimited by special "end of text" tokens to indicate new document boundaries.

The core task is language modeling: for every position in the sequence, the Transformer neural network attempts to predict the next token based on all preceding tokens.

- For a given token (e.g., a "green cell" in a diagram), the Transformer processes all prior tokens (the "yellow tokens") as its context.
- The output is a probability distribution over the entire vocabulary, indicating the likelihood of each possible next token.
- This prediction is supervised: the actual next token in the sequence serves as the ground truth.
- A loss function (e.g., cross-entropy loss) is used to update the Transformer's weights, encouraging it to assign higher probabilities to the correct next token. This process happens in parallel across all cells and is iterated over many batches of data.

Initially, with random weights, the model produces incoherent outputs. As training progresses, the model learns to generate increasingly coherent and consistent text, eventually understanding elements like words, spaces, and punctuation.

8.2.5 Output of Pre-training: Base Models

The pre-training process, specifically the language modeling task, compels the Transformer to learn powerful general representations of language and underlying concepts. This means the model is forced to implicitly "multitask" a vast number of linguistic tasks just by predicting the next token.

Early Application: Fine-tuning (GPT-1 Era) Initially, a primary application of these pre-trained "base models" was efficient fine-tuning for specific downstream tasks, even with very few examples. For instance, instead of training a sentiment classification model from scratch, one could fine-tune a pre-trained Transformer.

Evolution: Prompt Engineering (GPT-2 Era) It was later discovered that pre-trained base models could be effectively "prompted" without any further fine-tuning. Since these models are designed to complete documents, they can be "tricked" into performing tasks by structuring prompts that imitate a desired document format. For example, a few-shot prompt (providing a few Q&A examples before posing a new question) can elicit an answer as the model attempts to complete the document. This technique became known as "prompt engineering" and demonstrated that powerful results could be achieved without neural network training or fine-tuning.

Base Models vs. Assistant Models It is crucial to distinguish between base models and "assistant models".

- **Base Models**: These are document completers, not designed to answer questions or be conversational assistants. If asked to "write a poem about bread and cheese," a base model might respond with more questions or just continue the pattern it thinks it's completing. While it is possible to craft specific few-shot prompts to make a base model behave somewhat like an assistant, this approach is generally unreliable.
- **Assistant Models**: These are specifically trained (via subsequent stages) to be helpful assistants. ChatGPT, for instance, is an assistant model.

Examples of base models include GPT-3 (available via API as DaVinci) and GPT-2 (weights available on GitHub). The GPT-4 base model was never publicly released. The LLaMA series from Meta is considered one of the best available base models currently, though it lacks commercial licensing.

8.3 Stage 2: Supervised Fine-tuning (SFT Model)

To create actual GPT assistants that reliably respond to queries, supervised fine-tuning is employed.

8.3.1 Data Collection for SFT

This stage involves collecting small but high-quality datasets consisting of "prompt and ideal response" pairs. Human contractors are typically hired to gather this data, generating ideal responses to prompts. These contractors follow extensive labeling documentation, instructing them to be helpful, truthful, and harmless. Tens of thousands of such pairs are typically collected.

8.3.2 Algorithm for SFT

The algorithmic approach remains language modeling, identical to pre-training. However, the training set is swapped from large quantities of general internet documents to lower quantities of high-quality, task-specific prompt-response data.

8.3.3 Output of SFT

The result is an "SFT model," which can be deployed as a functional assistant. These models work to some extent in answering queries directly.

8.4 Stages 3 & 4: Reinforcement Learning from Human Feedback (RLHF)

RLHF is a further refinement pipeline that consists of two sub-stages: Reward Modeling and Reinforcement Learning.

8.4.1 Why RLHF?

RLHF is generally employed because it significantly improves model performance beyond SFT. Human evaluation consistently shows a preference for models trained with RLHF (e.g., PPO models) over SFT models or base models prompted to act as assistants.

One hypothesis for this improvement lies in the asymmetry between generation and comparison. It is often much easier for humans to judge the quality of several generated outputs than to meticulously craft a single ideal output. This allows for more effective leveraging of human judgment.

However, RLHF models are not strictly superior in all cases; they may lose some "entropy" compared to base models, leading to less diverse outputs. Base models, with their higher entropy, can be better suited for tasks requiring diverse suggestions (e.g., generating many unique Pokemon names).

8.4.2 Stage 3: Reward Modeling (RM)

- **Data Collection for RM**: The data for this stage consists of comparisons. The SFT model is used to generate multiple completions (e.g., three) for a given prompt. Human contractors then rank these completions based on quality. This ranking process can be very difficult and time-consuming for humans.
- **Algorithm for RM**: This is framed as a classification task where the model learns to predict a "reward" score for a completion given a prompt.
  - The prompt and each completion are laid out as input.
  - A special "reward readout token" is appended to the completion.

  - The Transformer is then supervised only at this single token position to predict a numerical reward value representing the completion's quality.
  - A loss function enforces consistency between the model's predicted rewards and the human-provided rankings.

- **Output of RM**: A "reward model" that can score the quality of any arbitrary completion for any given prompt. This model is not deployable as an assistant itself but is crucial for the subsequent RL stage.

8.4.3 Stage 4: Reinforcement Learning (RL)

- **Process**: In this stage, a large collection of prompts is used. The assistant model (initialized from the SFT model) generates completions for these prompts.
  - For each completion, the pre-trained and fixed reward model assigns a quality score.
  - The standard language modeling loss function is applied to the tokens generated by the assistant model.
  - Critically, this language modeling objective is weighted by the rewards provided by the reward model.
  - Completions that receive high reward scores will have their constituent tokens reinforced (given higher probabilities for future generation). Conversely, tokens from low-scoring completions will have their probabilities reduced.
  - This process is repeated across many prompts and batches, iteratively improving the assistant model's ability to generate high-quality outputs according to the reward model.
- **Output of RL**: A "policy" (the trained model) that consistently generates completions scoring high according to the reward model.

8.4.4 Examples of Models and Their Training Stages

- **RLHF Models**: ChatGPT, GPT-4, Claude, GPT-3.5. These are typically preferred by humans.
- **SFT Models**: Vicuna 13B, Koala.
- **Base Models**: GPT-3 (DaVinci), LLaMA series.

Model performance rankings (e.g., Elo ratings) show GPT-4 as the most capable, followed by Claude and GPT-3.5.

8.5 Applying GPT Assistants Effectively

Effective application of GPT assistants requires understanding their inherent "cognitive" differences from human thought processes and leveraging prompt engineering and external tools to compensate.

8.5.1 Understanding the GPT "Cognitive" Model

A human's internal thought process for generating a sentence involves a rich internal monologue, tool use (e.g., Wikipedia, calculator), reflection, sanity checks, self-correction, and iterative refinement. In contrast, a GPT model is fundamentally a "token simulator".

- GPTs process and generate text token by token, spending roughly the same amount of computational work per token.
- Unlike humans, the training data for LLMs strips away all internal dialogue, so the models do not inherently reflect, sanity check, or correct their mistakes along the way. They simply try their best to imitate the next token.
- **GPT Cognitive Advantages**:
  - **Vast Factual Knowledge**: Possess a very large fact-based knowledge across numerous areas due to billions of parameters, acting as extensive storage for facts.
  - **Large, Perfect Working Memory (Context Window)**: Whatever fits into the context window is immediately and losslessly accessible to the Transformer through its internal self-attention mechanism. This is effectively a perfect, albeit finite, memory.

Prompting, to a large extent, serves to bridge this "cognitive difference" between human brains and LLM "brains".

8.5.2 Prompt Engineering Techniques for Enhanced Performance

These techniques are designed to elicit more deliberate and effective responses from LLMs.

Spreading Out Reasoning ("Tokens to Think") LLMs need "tokens to think". They cannot perform complex reasoning per token and require the reasoning process to be spread out across more tokens.

- **Show Your Work (Few-shot Prompting)**: Provide examples in the prompt that demonstrate a step-by-step reasoning process. The Transformer will imitate this template, leading to better results.
- **"Let's think step by step"**: Adding this phrase to a prompt can condition the Transformer to show its work. This forces it to perform less computational work per token, resulting in slower but more successful reasoning.

Self-Consistency LLMs can "get unlucky" and sample a suboptimal token, leading them down a "blind alley" in reasoning from which they cannot recover by default.

- To mitigate this, sample multiple times for a given prompt.
- Implement a process to identify and keep the best samples, or use a majority vote among the outputs.

Self-Correction / Reflection Surprisingly, larger LLMs (like GPT-4) can often “know” when they have made a mistake, even if they don’t correct it by default.

- Explicitly ask the model: “Did you meet the assignment?”.
- For example, if asked to generate a non-rhyming poem that ends up rhyming, GPT-4 might admit it failed when prompted to check. Without this explicit instruction, it will not self-check.

Recreating “System 2” Thinking Drawing a parallel to human cognition (System 1: fast, automatic; System 2: slower, deliberate planning), many prompt engineering techniques aim to induce “System 2” behavior in LLMs.

- **Tree of Thought**: A recently developed technique that involves maintaining multiple completions for a given prompt and scoring them along the way, keeping only the promising paths. This is not just a simple prompt but requires “python glue code” to manage the tree search algorithm. This is analogous to Monte Carlo Tree Search used in systems like AlphaGo.
- **ReAct (Reasoning and Acting)**: A method where the LLM’s answer is structured as a sequence of “thought, action, observation”. This allows the model to perform a full “rollout” of its thinking process and permits “tool use” within its actions.
- **AutoGPT**: An inspirational project that enables an LLM to maintain a task list and recursively break down tasks. While not yet practical for real-world applications, it indicates the direction of research towards more autonomous agents.

8.5.3 Asking for Success: Conditioning on High Performance

LLMs are trained on datasets containing a wide spectrum of quality (e.g., wrong student answers alongside expert solutions). By default, they aim to imitate all of this. To obtain high-quality output:

- **Explicitly Ask for Quality**: Use phrases that condition the Transformer to aim for correctness and high performance. For example, “Let’s work this out in a step-by-step way to be sure we have the right answer” has been shown to be very effective. This prevents the Transformer from hedging its probability mass on low-quality solutions.
- **Role-playing**: Instruct the LLM to adopt a persona: “You are a leading expert on this topic,” or “pretend you have IQ 120”. However, be cautious not to demand excessively high “IQ” (e.g., 400), as this might push the model out of its data distribution or into sci-fi/role-playing modes.

8.5.4 Tool Use for LLMs

Just as humans rely on tools for tasks they aren’t good at (e.g., calculators for complex arithmetic), LLMs can benefit from tool integration.

- LLMs may not inherently “know” their weaknesses.
- **Explicit Instructions**: Tell the Transformer when and how to use external tools. For instance, “You are not very good at mental arithmetic. Whenever you need to do very large number addition/multiplication, instead use this calculator. Here’s how you use the calculator: [token combination]”.
- Tools can include calculators, code interpreters, and search engines.

8.5.5 Retrieval Augmented Generation (RAG)

This approach combines the strengths of LLMs’ extensive internal memory with external, retrieved information.

- The context window of a Transformer acts as its perfect working memory. If relevant information is loaded into this memory, the model performs exceptionally well.
- **RAG Recipe**:
  1. Gather relevant documents.
  2. Split documents into smaller chunks.
  3. Embed all chunks into vector representations (embedding vectors).
  4. Store these vectors in a vector store (e.g., a vector database).
  5. At query time, use the user’s query to fetch the most relevant chunks from the vector store.
  6. “Stuff” these retrieved chunks into the LLM’s prompt as additional context.
  7. The LLM then generates a response using both its internal knowledge and the provided context.
- This mirrors how humans consult primary documents or textbooks when solving problems, rather than relying solely on memory. LlamaIndex is mentioned as a data connector for this purpose.

8.5.6 Constraint Prompting

These techniques force LLMs to produce outputs in a specific, desired format.

- **Guidance (Microsoft)**: An example of a library that enforces specific output structures, such as JSON. This is achieved by manipulating the probabilities of tokens during generation, allowing the Transformer to fill in the blanks while adhering to the specified form and potentially additional restrictions.

8.5.7 Fine-tuning Models (Changing Weights)

While prompt engineering is powerful, fine-tuning offers another level of customization by directly changing the model’s weights. This process has become more accessible recently.

- **Parameter Efficient Fine-tuning (PEFT)**: Techniques like LoRA (Low-Rank Adaptation) allow for fine-tuning only small, sparse parts of the model while keeping most of the base model's weights fixed. This significantly reduces the cost and computational requirements of fine-tuning and allows for low-precision inference on the clamped parts.
- **Open-Source Base Models**: The availability of high-quality open-source base models (e.g., LLaMA, though not commercially licensed) facilitates fine-tuning.
- **Considerations for Fine-tuning**:
  - **Technical Involvement**: Fine-tuning is technically more involved and requires greater expertise.
  - **Data Requirements**: It necessitates human data contractors for custom datasets or complex synthetic data pipelines.
  - **Iteration Cycle**: It significantly slows down the development and iteration cycle.
  - **SFT vs. RLHF Complexity**:
    * **Supervised Fine-tuning (SFT)**: Relatively straightforward, as it's essentially a continuation of the language modeling task.
    * **Reinforcement Learning from Human Feedback (RLHF)**: Considered "very much research territory," highly unstable, difficult to train, and not beginner-friendly. It is also still evolving rapidly. It is generally not advised for individuals to attempt rolling their own RLHF implementations.

8.6 Recommendations for Achieving and Optimizing LLM Performance

8.6.1 Achieve Top Performance (Primary Focus)

1. **Use the Most Capable Model**: Currently, this is GPT-4.
2. **Detailed and Informative Prompts**: Craft prompts with abundant task content, relevant information, and clear instructions. Think of it as providing instructions to a human contractor who cannot ask follow-up questions.
3. **Cater to LLM Psychology**: Remember that LLMs lack human qualities like inner monologue or inherent cleverness. Structure prompts to compensate for these differences.
4. **Retrieve and Add Context (RAG)**: Always provide any relevant external information to the prompt. Explore various prompt engineering techniques available online.
5. **Experiment with Few-Shot Examples**: Show, don't just tell. Providing examples helps the model understand the desired output format and behavior.
6. **Utilize Tools and Plugins**: Offload tasks that LLMs struggle with natively (e.g., calculations, code execution, search) to external tools.

7. **Implement Chains and Reflection**: Move beyond single prompt-answer interactions. Consider sequences of prompts, reflection steps, and generating multiple samples to select the best one.

8.6.2 Optimize Performance (Secondary Focus)

1. **Consider Fine-tuning (After Prompt Engineering)**: Once you've exhausted the capabilities of prompt engineering, investigate fine-tuning your model. Be prepared for a slower and more involved process.
2. **RLHF (Expert/Research Zone)**: While RLHF can yield better results than SFT, it is highly complex and difficult to implement successfully. It remains a research area.
3. **Cost Optimization**: To manage costs, explore using lower-capacity models or developing shorter, more efficient prompts.

8.7 Use Cases and Limitations of Current LLMs

8.7.1 Current Limitations

It is critical to be aware of the significant limitations of LLMs today:

- **Bias**: Models may exhibit biases present in their training data.
- **Fabrication/Hallucination**: LLMs can generate factually incorrect or nonsensical information.
- **Reasoning Errors**: They may make logical mistakes or struggle with complex reasoning.
- **Application-Specific Struggles**: They may perform poorly in entire classes of applications.
- **Knowledge Cutoffs**: Their knowledge is limited to their training data and does not include information after a certain date (e.g., September 2021 for some models).
- **Vulnerability to Attacks**: LLMs are susceptible to various attacks, including prompt injection, jailbreak attacks (bypassing safety guardrails), and data poisoning.

8.7.2 Recommended Use Cases

Given the current limitations, it is advised to use LLMs in specific ways:

- **Low-Stakes Applications**: Employ LLMs where the consequences of errors are minimal.
- **Human Oversight**: Always combine LLM usage with human review and oversight.
- **Source of Inspiration/Suggestions**: Leverage them for brainstorming, generating ideas, or providing starting points.

- **Co-pilots, Not Autonomous Agents**: View LLMs as assistive tools rather than fully autonomous agents performing tasks independently. The models are not yet at a stage for complete autonomy.

8.8 Conclusion

GPT-4 is an impressive technological artifact with vast knowledge across many domains, capable of tasks like math and coding. A vibrant ecosystem of tools and applications is being built around these models, making their power increasingly accessible.

The following Python code snippet demonstrates the simplicity of interacting with a GPT-4 assistant model via an API call:

```python
import openai

# Replace with your actual API key
openai.api_key = "YOUR_API_KEY"

response = openai.chat.completions.create(
    model="gpt-4",  # Or another model like "gpt-3.5-turbo"
    messages=[
        {"role": "user", "content": "Can you say something to inspire the audience of Microsoft Build 2023?"}
    ]
)

print(response.choices[0].message.content)
```

Listing 79: Basic GPT-4 API Interaction

This simple interaction allows users to leverage the capabilities of GPT-4 and receive a high-quality, inspiring response. The ecosystem surrounding these models is continuously evolving, offering powerful new possibilities for developers and users alike.