

OFFICIAL COURSE STUDY GUIDE

Neural Networks: Zero to Hero

Build neural networks from scratch in code, with Andrej Karpathy.

AI & AGENTIC ENGINEERING · ADVANCED · 3 MODULES · 19 HR 29 MIN

Course by Andrej Karpathy · Curated and published by MegaCourse

MEGACOURSE.APP

Neural Networks: Zero to Hero: The Complete Study Guide

2026 Edition

Published by MegaCourse · megacourse.app

Learn Anything. Achieve Everything.

MegaCourse curates the world's best free courses into structured learning paths with study guides, practice material, and certificates. This guide accompanies the MegaCourse course "Neural Networks: Zero to Hero", taught by Andrej Karpathy. It is designed to be read alongside the video lessons and to serve as a standalone reference for review.

Study guide text copyright © 2026 MegaCourse. Course video content remains the property of its creators. You may print this guide for personal study. Redistribution or resale is not permitted.

Enroll, track progress, and earn your certificate at
megacourse.app/courses/neural-networks-zero-to-hero

Contents

<i>About This Course</i>	<i>4</i>
CHAPTER 1 Neural Networks and Language Models from Scratch	6
CHAPTER 2 Code, Notes and Prerequisites	15
CHAPTER 3 Final Assessment: Neural Nets	20
<i>Glossary</i>	<i>23</i>
<i>Answer Key</i>	<i>25</i>
<i>Where to Go Next</i>	<i>28</i>

INTRODUCTION

About This Course

Welcome to the official study guide for "Neural Networks: Zero to Hero." This MegaCourse offers a structured learning path to build neural networks from first principles. You will code these systems from scratch alongside Andrej Karpathy, gaining a deep understanding of their inner workings.

This course focuses on practical implementation. You will move beyond high-level frameworks to truly understand the mechanics of modern AI models. We cover everything from basic neural network construction to implementing backpropagation by hand and building a GPT-style language model.

MegaCourse has transformed the original video series into a complete educational experience. This includes a clear structure, curated reading materials, additional resources, and a final knowledge check quiz. This guide provides a written companion to the video lessons, reinforcing key concepts and offering a reference for your studies.

Course Facts

FORMAT	3 modules · 13 lessons
VIDEO INSTRUCTION	About 19 hr 29 min
LEVEL	Advanced
CATEGORY	AI & Agentic Engineering
INSTRUCTOR	Andrej Karpathy
CERTIFICATE	Included with course completion on MegaCourse
COURSE HOME	megacourse.app/courses/neural-networks-zero-to-hero

What You Will Learn

- Build and train neural nets from first principles.
- Implement backpropagation by hand (micrograd).
- Code a GPT-style language model from scratch.
- Truly understand what happens inside an LLM.

Who This Course Is For

This "Neural Networks: Zero to Hero" course is designed for advanced students and practitioners in AI and agentic engineering. It targets individuals who want to move past using existing libraries and

understand the fundamental computations within neural networks.

To succeed, you should arrive with a solid grasp of Python programming. Basic calculus knowledge is also necessary, particularly concepts related to derivatives. Some prior exposure to machine learning concepts will be beneficial, though not strictly required for understanding the core material.

Before You Start

- Solid Python programming skills.
- Basic calculus knowledge.
- Some familiarity with machine learning concepts.

How to Use This Guide

This study guide serves as your comprehensive companion to the "Neural Networks: Zero to Hero" video lessons. We recommend watching each video lesson first to absorb Andrej Karpathy's explanations and coding demonstrations. Afterwards, use this guide to review the concepts, solidify your understanding, and refer to key code snippets or theoretical points.

The course is organized into 3 modules and 13 lessons, totaling approximately 19 hours and 29 minutes of instruction. You can pace your learning according to your schedule, but consistent engagement is key. Break down the material into manageable segments.

This guide also provides a reference for the curated reading and resource lessons. Use these to broaden your perspective and explore related topics. The final knowledge-check quiz will help you assess your comprehension of the entire course material.

CHAPTER 1

Neural Networks and Language Models from Scratch

This chapter establishes the foundational principles of neural networks and their application to language modeling. We begin by building a minimal automatic differentiation engine, a crucial component for training any neural network. This allows us to understand how gradients, which guide learning, are computed through complex mathematical expressions.

From these first principles, we transition to building character-level language models. We explore various architectures, starting with simple statistical models and progressing to multi-layer perceptrons (MLPs) and convolutional networks. The chapter culminates in constructing a simplified version of the Generative Pre-trained Transformer (GPT), demonstrating the core mechanisms that enable modern large language models.

The Foundation: Building Micrograd and Backpropagation

Covers: The spelled-out intro to neural networks and backpropagation: building micrograd (video, 2 hr 26 min)

Understanding how neural networks learn requires a grasp of automatic differentiation, often called autograd. This process efficiently computes the gradients of a loss function with respect to the network's parameters. Gradients indicate the direction and magnitude to adjust parameters to reduce error. This section introduces `micrograd`, a simplified autograd engine, built from scratch.

The Value Object and Computational Graph

At the core of `micrograd` is the `Value` object. This object wraps a scalar number and tracks the operation that produced it. When you perform an arithmetic operation (addition, multiplication, exponentiation, `tanh`), a new `Value` object is created. This object retains references to its input `Value` objects and the specific operation performed. This forms a directed acyclic graph, known as the computational graph. Each node in this graph is a `Value` object, and edges represent operations.

```
class Value:
    def __init__(self, data, _children=(), _op=''):
        self.data = data
        self.grad = 0.0
        self._backward = lambda: None # Placeholder for backprop function
        self._prev = set(_children)
        self._op = _op

    def __add__(self, other):
        other = other if isinstance(other, Value) else Value(other)
        out = Value(self.data + other.data, (self, other), '+')
        def _backward():
```

```
other.grad += 1.0 * out.grad
out._backward = _backward
return out

# Similar methods for __mul__, __pow__, tanh, etc.
```

The Chain Rule and Backpropagation

To compute gradients, `micrograd` uses the chain rule from calculus. The chain rule states that if a variable `z` depends on `y`, and `y` depends on `x`, then the rate of change of `z` with respect to `x` is the product of the rate of change of `z` with respect to `y` and the rate of change of `y` with respect to `x`. In the computational graph, this means we can propagate gradients backward from the output node to the input nodes.

Backpropagation is an algorithm that applies the chain rule systematically. It starts by assigning a gradient of 1.0 to the final output `Value` (representing $dL/dL = 1$). Then, it traverses the computational graph in reverse topological order. For each `Value` object, its `_backward` method is called. This method distributes the `grad` received from its output to its input `Value` objects, according to the derivative of the operation that created it. For instance, in addition, the gradient is simply passed through to both inputs. In multiplication, the gradient is scaled by the other input's data.

```
def backward(self):
    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
    for node in reversed(topo):
        node._backward()
```

This process ensures that each `Value` object accumulates the correct gradient, $dL/d(\text{Value})$, which is the sum of all paths from the loss `L` to that `Value`. Understanding this manual implementation provides deep insight into how modern deep learning frameworks calculate gradients for complex models.

Character-Level Language Modeling with Makemore

Covers: The spelled-out intro to language modeling: building makemore (video, 1 hr 58 min) · Building makemore Part 2: MLP (video, 1 hr 16 min)

Language modeling involves predicting the next token (character or word) in a sequence given the preceding tokens. This section introduces `makemore`, a framework for building character-level language models, starting with a simple statistical approach and progressing to neural networks.

Bigram Language Model

The simplest language model is a bigram model. It predicts the next character solely based on the *current* character. To build this, we scan a large text corpus and count the occurrences of every character pair. For example, if 'h' is followed by 'e' 100 times and 'a' 5 times, then $P(e|h)$ is much higher than $P(a|h)$.

We store these counts in a 2D array or dictionary. To generate text, we start with a special 'start' token, sample the next character based on its probabilities, then use that character to sample the next, and so on, until an 'end' token is generated. A common issue is encountering unseen bigrams. To address this, we use smoothing techniques, such as adding a small count (e.g., 1) to all possible bigrams, ensuring no probability is zero.

```
# Example: Building a bigram count table
# N is a 27x27 matrix (26 letters + 1 start/end token)
# N[char1_idx, char2_idx] stores count of (char1, char2)

# Example of probability calculation (after smoothing)
# P = (N + 1) / (N.sum(axis=1, keepdims=True) + 27) # Add-1 smoothing
```

While simple, bigram models quickly reveal their limitations. They have a very short memory (only one preceding character) and cannot capture complex dependencies or long-range patterns in language. This motivates the move to neural networks.

Transition to Multi-Layer Perceptrons (MLPs)

To overcome the limitations of n-gram models, we introduce Multi-Layer Perceptrons (MLPs). An MLP can learn more complex, non-linear relationships between input and output. For character-level modeling, the input to the MLP is typically an *embedding* of the preceding characters. Each character is mapped to a dense vector representation (an embedding), which the network learns during training. These embeddings capture semantic similarities between characters.

For `makemore`, we might take a context of, say, three preceding characters. Each character's embedding is concatenated to form a single input vector for the MLP. The MLP then processes this vector through several layers, each consisting of a linear transformation (matrix multiplication plus bias) followed by a non-linear activation function. The final layer outputs logits for each possible next character. These logits are converted into probabilities using a softmax function, and a negative log likelihood loss is computed against the true next character. This setup allows the model to consider a wider context and learn richer patterns than a simple bigram model.

Makemore: Activations, Gradients, and Normalization

Covers: Building makemore Part 3: Activations & Gradients, BatchNorm (video, 1 hr 56 min) · Building makemore Part 4: Becoming a Backprop Ninja (video, 1 hr 55 min)

As we build deeper neural networks for `makemore`, several challenges arise, particularly concerning the stability of training. This section explores activation functions, gradient dynamics, and Batch Normalization as solutions.

Activation Functions and Gradient Flow

Linear layers alone, stacked one after another, only produce another linear function. To model complex, non-linear relationships, neural networks require non-linear activation functions. Common choices include `tanh` and `ReLU` (Rectified Linear Unit).

- **tanh (Hyperbolic Tangent):** Squashes values between -1 and 1. Its derivative is $1 - \tanh(x)^2$. As x moves far from zero, $\tanh(x)$ approaches 1 or -1, and its derivative approaches 0. This can lead to *vanishing gradients*, where gradients become tiny in early layers, slowing or stopping learning.
- **ReLU:** Outputs x for positive inputs and 0 for negative inputs. Its derivative is 1 for positive inputs and 0 for negative inputs. `ReLU` helps mitigate vanishing gradients for positive activations but can suffer from the 'dying ReLU' problem if neurons consistently output zero.

Understanding how gradients flow through these activations is crucial for manual backpropagation. For `tanh`, the gradient $d(\text{out})/d(\text{in})$ is $1 - \text{out.data}^2$. For `ReLU`, it is 1 if $\text{in.data} > 0$ and 0 otherwise.

Weight Initialization and Gradient Issues

Proper initialization of network weights is critical. If weights are too large, activations can saturate (`tanh`) or explode. If too small, activations can vanish. A common strategy is to initialize weights from a small random distribution, often scaled by the inverse square root of the number of inputs (Kaiming or Xavier initialization). This helps maintain a reasonable variance of activations and gradients throughout the network.

Batch Normalization

One significant challenge in training deep networks is *internal covariate shift*, where the distribution of activations in intermediate layers changes during training as parameters of previous layers are updated. This forces subsequent layers to constantly adapt, slowing down learning. Batch Normalization (BatchNorm) addresses this by normalizing the inputs to each layer.

For a given batch of inputs x , BatchNorm performs the following steps:

1. Calculate the mean μ_B and variance σ_B^2 of x across the batch dimension.
2. Normalize x : $x_{\text{hat}} = (x - \mu_B) / \sqrt{\sigma_B^2 + \epsilon}$, where ϵ is a small constant for numerical stability.
3. Scale and shift: $y = \gamma * x_{\text{hat}} + \beta$, where γ (scale) and β (shift) are learnable parameters. These parameters allow the network to restore the original distribution if normalization is detrimental.

```
# Conceptual BatchNorm forward pass
# x: input tensor (batch_size, num_features)
mean = x.mean(dim=0, keepdim=True)
var = x.var(dim=0, keepdim=True)
x_hat = (x - mean) / (var + 1e-5).sqrt()
out = gamma * x_hat + beta
```

During inference, the batch statistics (μ_B , σ_B^2) are replaced by moving averages computed during training. BatchNorm significantly stabilizes training, allows for higher learning rates, and reduces the

sensitivity to initialization. Manually implementing the backpropagation for BatchNorm, though complex, reinforces understanding of how gradients flow through normalization layers.

Makemore: Building a WaveNet-style Model

Covers: Building makemore Part 5: Building a WaveNet (video, 56 min)

While MLPs improve upon n-gram models, they still face limitations, particularly with capturing long-range dependencies efficiently. Standard MLPs process a fixed-size context window. To expand the receptive field without a proportional increase in parameters, we can utilize convolutional neural networks (CNNs), specifically with dilated convolutions.

Convolutional Neural Networks for Sequences

Traditional CNNs are often associated with image processing, where they apply filters across spatial dimensions. For sequence data, like characters in a name, we use 1D convolutions. A 1D convolutional layer applies a filter (a small weight matrix) across a sequence, performing dot products at each position. This allows the network to detect local patterns (e.g., character combinations) anywhere in the input sequence, benefiting from parameter sharing.

```
# Conceptual 1D Convolution
# input_sequence: [batch_size, sequence_length, embedding_dim]
# filter: [filter_size, embedding_dim, output_channels]
# output: [batch_size, new_sequence_length, output_channels]

def conv1d(input, filter, stride=1):
    # Apply filter across input with given stride
    pass
```

Dilated Convolutions

Dilated convolutions are a powerful variant that allows the convolutional filter to skip input values with a certain step or 'dilation rate'. Instead of processing contiguous elements, a dilated convolution with a dilation rate d processes elements that are d positions apart. This effectively expands the *receptive field* of the filter without increasing the number of parameters or losing resolution (as pooling layers do).

For example, a filter of size 2 with a dilation rate of 1 processes two adjacent elements. With a dilation rate of 2, it processes two elements separated by one element. This allows a small filter to 'see' a much larger portion of the input sequence. By stacking layers with exponentially increasing dilation rates (e.g., 1, 2, 4, 8), a WaveNet-style model can achieve a very large receptive field, enabling it to capture long-range dependencies in the sequence data, which is critical for generating coherent names or other sequences.

Building a WaveNet for Makemore

In the context of `makemore`, a WaveNet-style architecture would take a sequence of character embeddings as input. It would then pass this through several layers of 1D dilated convolutions, potentially followed by non-linear activations and residual connections. Each layer would progressively increase the effective context window. The final layer would typically output logits for the next character, similar to the MLP

approach, which are then converted to probabilities via softmax. This architecture allows the model to learn more sophisticated character generation patterns, using a broader historical context than previous models.

Building GPT: Architecture and Tokenization

Covers: Let's build GPT: from scratch, in code, spelled out. (video, 1 hr 56 min) · State of GPT | BRK216HFS (video, 43 min) · Let's build the GPT Tokenizer (video, 2 hr 14 min)

The Generative Pre-trained Transformer (GPT) architecture represents a significant advancement in language modeling, moving beyond character-level predictions to token-level understanding. This section covers the core components of GPT and the essential preprocessing step of tokenization.

The Transformer Architecture: Decoder-Only

GPT models are based on the Transformer architecture, specifically a *decoder-only* variant. Unlike the original Transformer, which had both an encoder and a decoder, GPT focuses solely on generating sequences. Its primary mechanism is *self-attention*.

Self-attention allows each token in an input sequence to weigh the importance of every other token in the same sequence. For a given token, it computes three vectors: Query (Q), Key (K), and Value (V). The attention score between two tokens is calculated by taking the dot product of their Query and Key vectors. These scores are then scaled, passed through a softmax function to get probabilities, and multiplied by the Value vectors. The sum of these weighted Value vectors forms the output for that token.

```
# Conceptual Self-Attention
Q = input_embedding @ W_query
K = input_embedding @ W_key
V = input_embedding @ W_value

attention_scores = (Q @ K.transpose(-2, -1)) / (K.shape[-1]**0.5)
masked_attention_scores = apply_causal_mask(attention_scores) # For GPT
attention_weights = softmax(masked_attention_scores)
output = attention_weights @ V
```

GPT uses *masked self-attention* to ensure causality: when predicting the next token, a token can only attend to preceding tokens, not future ones. This is achieved by setting attention scores for future tokens to negative infinity before the softmax, effectively zeroing out their contribution. GPT also employs *multi-head attention*, where multiple self-attention mechanisms run in parallel, allowing the model to focus on different aspects of the input simultaneously. Positional encodings are added to the input embeddings to provide information about the token's position in the sequence, as self-attention itself is permutation-invariant. Each Transformer block also includes a feed-forward neural network, residual connections, and layer normalization to aid training stability.

The State of GPT and Its Capabilities

GPT models, particularly large ones, demonstrate remarkable capabilities in generating human-like text, translating languages, answering questions, and performing various other natural language tasks. Their

success stems from their ability to learn complex patterns and relationships in vast amounts of text data during pre-training, followed by fine-tuning for specific tasks. The *State of GPT* discussion highlights the evolution, scale, and emergent properties of these models, emphasizing their general-purpose language understanding and generation abilities.

Building the GPT Tokenizer

Before text can be fed into a GPT model, it must be converted into numerical tokens. This process is called tokenization. GPT models typically use a Byte-Pair Encoding (BPE) tokenizer. BPE works by iteratively merging the most frequent adjacent pairs of characters or character sequences in a training corpus. It starts with a vocabulary of individual characters and builds up a vocabulary of common words and subword units.

For example, if 'e' and 's' are frequent, they might merge to 'es'. If 'es' and 't' are frequent, they might merge to 'est'. This creates a vocabulary that balances between character-level (too long sequences) and word-level (too large vocabulary, out-of-vocabulary words) tokenization. BPE ensures that common words are represented by single tokens, while rare words are broken down into subword units that are still present in the vocabulary. This makes the tokenizer efficient and robust to unseen words. The tokenizer also handles special tokens for start-of-sequence, end-of-sequence, and padding.

Reproducing GPT-2 (124M)

Covers: Let's reproduce GPT-2 (124M) (video, 4 hr 1 min)

After understanding the core components of `micrograd`, `makemore`, and the GPT architecture, the final step in this module is to bring it all together by reproducing a small version of GPT-2. This involves implementing the full training and inference pipeline for a model with 124 million parameters, providing practical experience with a significant deep learning project.

Scaling the Architecture

Reproducing GPT-2 involves scaling up the concepts learned in the previous sections. The model will consist of multiple Transformer decoder blocks, each with multi-head masked self-attention, feed-forward networks, residual connections, and layer normalization. The number of layers, attention heads, and embedding dimensions will be set to match the specifications of the original GPT-2 (124M) model. The output layer will be a linear layer mapping the final hidden states to the size of the tokenizer's vocabulary, followed by a softmax to produce next-token probabilities.

Data Preparation and Training Loop

Training a GPT-2 model requires a substantial dataset. For reproduction purposes, a smaller, representative dataset can be used to demonstrate the training process. The data must first be tokenized using the BPE tokenizer. The training loop involves:

1. **Data Loading:** Efficiently loading batches of tokenized text sequences.
2. **Forward Pass:** Feeding the input sequences through the GPT-2 model to obtain predicted logits for the next token.

3. **Loss Calculation:** Computing the cross-entropy loss between the predicted logits and the true next tokens in the sequence. The `micrograd` principles apply here to track the computational graph.
4. **Backward Pass:** Using backpropagation to compute gradients of the loss with respect to all model parameters. This is where the `micrograd` understanding becomes crucial, even if using a framework that handles it automatically.
5. **Parameter Update:** Adjusting the model's weights and biases using an optimizer, such as Adam, based on the computed gradients. A learning rate schedule is often employed to gradually decrease the learning rate over time.

```
# Simplified Training Loop Structure
optimizer = Adam(model.parameters(), lr=learning_rate)

for epoch in range(num_epochs):
    for batch in dataloader:
        inputs, targets = batch

        optimizer.zero_grad() # Clear gradients from previous step

        logits = model(inputs) # Forward pass
        loss = cross_entropy_loss(logits, targets) # Calculate loss

        loss.backward() # Backward pass, compute gradients
        optimizer.step() # Update parameters

    print(f"Epoch {epoch+1}, Loss: {loss.item()}")
```

Text Generation (Inference)

After training, the model can generate new text. This involves an iterative process:

1. Start with a prompt (e.g., a few initial tokens).
2. Feed the prompt into the trained model to get probability distributions for the next token.
3. Sample a token from this distribution (e.g., using greedy sampling or top-k/nucleus sampling).
4. Append the sampled token to the sequence.
5. Repeat steps 2-4 until a desired length is reached or an end-of-sequence token is generated.

Reproducing GPT-2 provides hands-on experience with the entire lifecycle of building and training a modern large language model, from data preparation to model inference, solidifying the theoretical concepts covered throughout the module.

KEY TAKEAWAYS

- Neural networks learn by adjusting parameters through backpropagation, which efficiently computes gradients using the chain rule on a computational graph.
- The `Value` object, as implemented in `micrograd`, tracks data, gradients, and the operations that created it, forming the basis for automatic differentiation.
- Character-level language models, starting with simple bigrams and progressing to MLPs, predict the next character in a sequence based on preceding context.
- Non-linear activation functions like `tanh` and `ReLU` are essential for neural networks to model complex relationships, but require careful handling of gradient flow.
- Batch Normalization stabilizes neural network training by normalizing layer inputs, mitigating internal covariate shift and allowing for faster convergence.
- The GPT architecture utilizes a decoder-only Transformer with masked self-attention to efficiently process and generate token sequences, capturing long-range dependencies.
- Tokenization, typically using Byte-Pair Encoding (BPE), converts raw text into numerical subword units, balancing vocabulary size and handling unseen words for language models.

CHECK YOUR UNDERSTANDING

Answer these before moving on. Model answers are in the Answer Key at the back of this guide.

1. Explain the purpose of a computational graph in the context of neural networks.
2. Describe the core idea behind backpropagation and how it uses the chain rule.
3. How does a bigram language model predict the next character? What are its limitations?
4. What problem does Batch Normalization address, and how does it work?
5. Briefly explain the key innovation of the Transformer architecture, specifically self-attention.
6. Why is tokenization a necessary step before training a language model?

CHAPTER 2

Code, Notes and Prerequisites

Welcome to Chapter 2 of the Neural Networks: Zero to Hero study guide. This chapter prepares you for the hands-on coding experience that defines this course. You will learn how to set up your development environment, access the companion code repositories, and adopt an effective learning methodology centered on active coding and experimentation.

Crucially, this chapter also reviews the foundational mathematical concepts essential for understanding neural networks. While you do not need an advanced degree, a solid grasp of derivatives, the chain rule, vectors, matrices, and basic probability is indispensable. We will explore why these concepts are critical and how they apply directly to building and training neural networks from scratch.

Setting Up Your Development Environment for Active Learning

Covers: Follow Along in Code

The Neural Networks: Zero to Hero course emphasizes a code-first approach. This means you learn by doing, not just by watching. To truly grasp the material, you must type out the code yourself, experiment with it, and even intentionally break it to understand its mechanics. This chapter guides you through setting up your workspace and adopting this active learning style.

Accessing Companion Repositories

All course code is hosted on GitHub in dedicated repositories. You will need a basic understanding of Git to clone these repositories to your local machine. If you are new to Git, a quick online tutorial on cloning repositories will be beneficial. The primary repositories for this course include:

- **nn-zero-to-hero:** This repository contains Jupyter notebooks corresponding to each lecture. These notebooks are your primary resource for following along with Andrej Karpathy's code demonstrations.
- **micrograd:** This is a standalone, tiny auto-differentiation engine that you will build from scratch in the first lecture. It provides a fundamental understanding of how gradients are computed automatically.
- **makemore:** This repository focuses on character-level language models. It is where you will implement neural networks to generate text one character at a time.
- **minGPT / nanoGPT:** These repositories contain minimal, clean implementations of the GPT architecture. They serve as excellent starting points for understanding and building large language models.

To begin, ensure you have Python 3 installed on your system. It is highly recommended to use a virtual environment (such as `venv` or `conda`) for each project to manage dependencies cleanly. Once Python is ready, you can clone a repository using the command line:

```
git clone https://github.com/karpathy/nn-zero-to-hero.git
cd nn-zero-to-hero
pip install -r requirements.txt
jupyter notebook
```

This sequence clones the main course repository, navigates into its directory, installs necessary Python packages, and then launches Jupyter Notebook, allowing you to open and run the `.ipynb` files.

Effective Study Methodology

Your learning success hinges on your active participation. Follow these steps to maximize your understanding:

1. **Watch and Pause:** Watch a segment of a lecture video. When Karpathy writes code or explains a concept, pause the video.
2. **Reproduce the Code:** Switch to your own Jupyter notebook or Python environment. Type out the code shown in the video. Do not copy-paste. Typing helps build muscle memory and forces you to process each line.
3. **Run and Verify:** Execute your code. Ensure it produces the expected output. If you encounter errors, debug them. This is a crucial part of the learning process.
4. **Complete Exercises:** Each video description includes exercises. These are designed to deepen your understanding and apply the concepts in slightly different contexts. Make sure to attempt and solve them.
5. **Break Things on Purpose:** This is a powerful learning technique. Change a hyperparameter, modify a function, or alter an architectural detail. Before running the modified code, predict what effect your change will have. Then, run it and observe the actual outcome. This builds strong intuition about how different components of a neural network interact. For example, change a learning rate from $1e-3$ to 1.0 and observe the training behavior. Or, remove a non-linearity and see how it affects model capacity.

By diligently following this active, code-first approach, you will move beyond superficial understanding to a deep, practical mastery of neural networks.

Essential Mathematical Foundations for Neural Networks

Covers: Math You Need

Neural networks, at their core, are mathematical constructs. While you do not need to be a theoretical mathematician, a comfortable understanding of specific mathematical concepts is vital. This section outlines the minimum math required and explains why each topic is fundamental to building neural networks from first principles. If any of these areas feel unfamiliar, dedicate time to review them. Resources like 3Blue1Brown's 'Essence of Calculus' and 'Essence of Linear Algebra' are excellent for building intuition.

Derivatives and the Chain Rule

A neural network is essentially a very large, complex function that maps inputs to outputs. Training a neural network involves adjusting its internal parameters (weights and biases) to minimize a 'loss' function. This minimization process relies heavily on derivatives.

A **derivative** measures the rate at which a function's output changes with respect to a change in its input. In the context of neural networks, derivatives tell us how much the loss changes when a specific parameter changes. This information is used to update parameters in the direction that reduces the loss. The collection of all partial derivatives of the loss function with respect to each parameter is called the **gradient**.

The **chain rule** is a fundamental calculus rule for differentiating composite functions. A composite function is a function within a function, such as $f(g(x))$. Its derivative is $f'(g(x)) * g'(x)$. Neural networks are inherently composite functions. They consist of multiple layers, where the output of one layer becomes the input to the next. Backpropagation, the algorithm used to train neural networks, is nothing more than the repeated application of the chain rule to efficiently compute the gradients of the loss with respect to every parameter in the network, from the output layer all the way back to the input layer.

Consider a simple function $y = (3x + 2)^2$. To find dy/dx using the chain rule, let $u = 3x + 2$. Then $y = u^2$. We find $dy/du = 2u$ and $du/dx = 3$. Applying the chain rule, $dy/dx = (dy/du) * (du/dx) = 2u * 3 = 2(3x + 2) * 3 = 6(3x + 2)$. This process of breaking down a complex function into simpler, nested functions is exactly how backpropagation works through layers.

Vectors and Matrices

Vectors and **matrices** are the language of linear algebra, which is indispensable for neural networks. Data in neural networks is almost always represented as vectors or matrices. For example, an input image might be flattened into a vector, or a batch of data points might be represented as a matrix where each row is a data sample.

Weights and biases, the trainable parameters of a neural network, are also stored as vectors and matrices. The core operation within a neural network layer is typically a matrix multiplication, where input data (a vector or matrix) is multiplied by a weight matrix, and a bias vector is added. This operation efficiently performs many linear transformations in parallel.

For instance, if you have an input vector x of size N and a weight matrix W of size $M \times N$, the output y will be $W * x$, resulting in a vector of size M . Understanding matrix multiplication, vector addition, and dot products is crucial for comprehending how information flows and is transformed within a neural network.

Basic Probability

Probability theory plays a key role, especially in the output layers of classification models and in defining loss functions. Two crucial concepts are the softmax function and cross-entropy loss.

The **softmax function** is commonly applied to the raw outputs (often called 'logits') of a neural network in a classification task. It converts a vector of arbitrary real numbers into a probability distribution. The

output values of softmax are all positive and sum up to 1, making them interpretable as probabilities for each class. For example, if a network outputs logits `[2.0, 1.0, 0.1]` for three classes, softmax would convert these into probabilities like `[0.7, 0.2, 0.1]`.

Cross-entropy loss is a standard loss function used for training classification models. It quantifies the difference between two probability distributions: the true probability distribution (e.g., a one-hot encoded vector representing the correct class) and the probability distribution predicted by the model (the output of the softmax function). Minimizing cross-entropy loss during training encourages the model to output high probabilities for the correct class and low probabilities for incorrect classes.

Understanding these concepts reinforces the key intuition: a neural network is a large, differentiable function whose parameters are nudged in the opposite direction of the gradient of a loss function during training. All other complexities, such as different layer types, attention mechanisms, or normalization techniques, build upon this fundamental idea.

KEY TAKEAWAYS

- Learning neural networks effectively requires a code-first, hands-on approach where you actively reproduce and experiment with code.
- The course provides several companion GitHub repositories containing notebooks for lectures, an auto-differentiation engine, and implementations of language models.
- A robust study method involves watching, pausing, coding, completing exercises, and intentionally breaking code to understand its behavior.
- Derivatives and the chain rule are essential for understanding backpropagation, which computes gradients to update network parameters.
- Vectors and matrices are fundamental for representing data, weights, and performing the linear transformations within neural network layers.
- Basic probability concepts like the softmax function and cross-entropy loss are critical for interpreting model outputs and defining effective loss functions in classification tasks.
- At its core, a neural network is a differentiable function trained by adjusting parameters based on the gradient of a loss function.

CHECK YOUR UNDERSTANDING

Answer these before moving on. Model answers are in the Answer Key at the back of this guide.

1. Why is a 'code-first' approach emphasized in this course, and what are two specific ways to implement it in your study routine?
2. What is the primary purpose of the `micrograd` repository, and how does it relate to the mathematical foundations reviewed in this chapter?
3. Explain how the chain rule is fundamental to the training process of neural networks.
4. How are vectors and matrices used to represent data and operations within a neural network?

5. Describe the roles of the softmax function and cross-entropy loss in a neural network designed for classification.

CHAPTER 3

Final Assessment: Neural Nets

This final assessment chapter serves as a comprehensive review of the Neural Networks: Zero to Hero course. It provides an opportunity to consolidate your understanding of building and training neural networks from first principles. The assessment evaluates your grasp of core concepts, from implementing backpropagation by hand to coding a GPT-style language model.

Consider this assessment a tool to gauge your mastery of the material. It helps you identify areas where your knowledge is strong and pinpoint any topics that might require further study. Approaching it strategically will reinforce your learning and prepare you for advanced applications.

What You Should Have Mastered

Covers: Final Assessment: Neural Nets

By this point in the course, you should have a solid understanding of the fundamental principles behind neural networks. You should be able to build and train these networks from first principles. This includes a deep comprehension of how data flows through a network and how parameters are updated.

A critical skill is the ability to implement backpropagation by hand. This was demonstrated through the `micrograd` library. You should understand the chain rule application and how gradients are computed for each parameter. Also, you should be able to code a GPT-style language model from scratch. This involves understanding the architecture, including self-attention mechanisms and multi-head attention. Ultimately, you should truly understand what happens inside an LLM, from tokenization to generating text.

Approaching the Final Assessment

Covers: Final Assessment: Neural Nets

Treat the final assessment as a comprehensive review of all course material. Before starting, take a moment to recall the major concepts covered in each module. Read each question carefully, paying close attention to keywords and specific requirements. Some questions may test your theoretical knowledge, while others might require you to recall implementation details from the `micrograd` library or the GPT model.

Do not rush. If you encounter a challenging question, try to break it down. Consider sketching out a small example or recalling the relevant code snippets. If you feel unsure about a particular topic, make a mental note to revisit that lesson or chapter after the assessment. This approach helps you maximize your learning from the assessment itself.

Common Pitfalls and How to Avoid Them

Covers: Final Assessment: Neural Nets

Several common mistakes can occur during a comprehensive assessment. One pitfall is misinterpreting the nuances of a question. Always double-check what is being asked. Another common error is forgetting specific details about backpropagation, such as the exact flow of gradients or the role of activation functions. Reviewing the `micrograd` implementation can solidify these details.

Students sometimes confuse components of the GPT architecture. For example, they might mix up the purpose of self-attention versus the feedforward layers. A quick review of the Transformer architecture diagrams can prevent this. Avoid guessing. If you are truly stuck, make an educated choice and then prioritize reviewing that topic. A systematic review of your notes and the course material before the assessment can significantly reduce these errors.

Interpreting Your Results

Covers: Final Assessment: Neural Nets

Your score on the final assessment provides valuable feedback on your learning progress. A high score indicates a strong grasp of the course material and its practical applications. This means you are well-prepared to apply these skills in future projects.

If your score is lower than you hoped, view it as an opportunity for targeted review. Do not be discouraged. The assessment is a diagnostic tool. Analyze which questions you answered incorrectly. Pinpoint the specific concepts or lessons related to those questions. Revisit those particular sections of the course, review the code, and practice the exercises. This focused approach will help you solidify your understanding and turn weaknesses into strengths.

KEY TAKEAWAYS

- The final assessment evaluates your comprehensive understanding of neural networks.
- It tests both theoretical concepts and practical implementation skills.
- Strategic preparation and careful question analysis improve performance.
- Identifying common mistakes helps in avoiding them during the assessment.
- Use your assessment results to guide further, targeted study and reinforce learning.

CHECK YOUR UNDERSTANDING

Answer these before moving on. Model answers are in the Answer Key at the back of this guide.

1. What is the primary purpose of a loss function in a neural network?
2. Briefly explain the role of backpropagation in training a neural network.

3. How does the `micrograd` library help in understanding neural network training?
4. What is the significance of the self-attention mechanism in a GPT-style language model?
5. Describe one key difference between a simple multi-layer perceptron and a Transformer architecture.

REFERENCE

Glossary

Activation function

A non-linear function applied to the output of a neuron or layer in a neural network. It introduces non-linearity, allowing the network to learn complex patterns. Examples include `tanh` and `ReLU`.

Backpropagation

An algorithm for efficiently computing the gradient of a loss function with respect to all parameters in a neural network. It applies the chain rule by traversing the computational graph backward from output to input.

Batch normalization

A technique used to stabilize and accelerate the training of deep neural networks by normalizing the inputs to each layer. It reduces internal covariate shift by re-centering and re-scaling layer inputs.

Chain rule

A calculus rule for finding the derivative of a composite function, crucial for backpropagation in neural networks.

Code-first approach

A learning methodology emphasizing hands-on coding and active experimentation as the primary means of understanding technical concepts.

Computational graph

A directed acyclic graph representing a sequence of mathematical operations, where nodes are variables (like `Value` objects) and edges are operations. It is fundamental for automatic differentiation.

Cross-entropy loss

A common loss function used in classification tasks that measures the difference between two probability distributions (predicted and true).

Derivative

A measure of how a function's output changes with respect to a change in its input, indicating the rate of change or slope.

Gradient

A vector containing all the partial derivatives of a multivariable function, indicating the direction of the steepest ascent.

Gradient descent

An iterative optimization algorithm used to minimize a function. It adjusts parameters in the direction opposite to the gradient of the function, moving towards a local minimum.

Loss function

A measure of how well a neural network performs its task, quantifying the error between predicted and actual outputs.

Micrograd

A custom, lightweight auto-differentiation engine built from scratch in the course to understand backpropagation.

Multi-layer perceptron (mlp)

A type of feedforward neural network consisting of at least three layers (an input layer, one or more hidden layers, and an output layer) where each layer is fully connected to the next.

Neural network

A computational model inspired by the human brain, designed to recognize patterns.

Self-attention

A mechanism in Transformer networks that allows each element in a sequence to weigh the importance of all other elements in the same sequence when computing its representation. It is central to how GPT processes context.

Softmax function

A function that converts a vector of real numbers into a probability distribution, where all values are positive and sum to one.

Tokenization

The process of converting raw text into a sequence of numerical tokens suitable for input into a neural network. Byte-Pair Encoding (BPE) is a common subword tokenization method used in large language models.

Transformer

A neural network architecture primarily based on the self-attention mechanism, designed for sequence-to-sequence tasks. GPT is a decoder-only variant of the Transformer.

Virtual environment

An isolated Python environment that allows you to manage dependencies for different projects without conflicts.

REFERENCE

Answer Key

Chapter 1: Neural Networks and Language Models from Scratch

1. Explain the purpose of a computational graph in the context of neural networks.

A computational graph visually represents the sequence of operations performed to compute an output from inputs. In neural networks, it tracks how parameters contribute to the final loss. This structure is crucial for efficiently applying the chain rule during backpropagation to calculate gradients for all parameters.

2. Describe the core idea behind backpropagation and how it uses the chain rule.

Backpropagation is an algorithm that computes gradients of the loss function with respect to network parameters by propagating gradients backward through the computational graph. It uses the chain rule, which dictates how derivatives of composite functions are calculated. Each node in the graph computes the gradient for its inputs based on the gradient it receives from its output and the derivative of its local operation.

3. How does a bigram language model predict the next character? What are its limitations?

A bigram language model predicts the next character based solely on the immediately preceding character. It learns probabilities of character pairs from a training corpus. Its main limitation is its very short memory, meaning it cannot capture longer-range dependencies or complex patterns in language, leading to less coherent or natural-sounding generated text.

4. What problem does Batch Normalization address, and how does it work?

Batch Normalization addresses the problem of internal covariate shift, where the distribution of activations in intermediate layers changes during training. It works by normalizing the inputs to each layer within a mini-batch, re-centering them to have zero mean and unit variance. Learnable scale and shift parameters (gamma and beta) then allow the network to adjust this normalized distribution.

5. Briefly explain the key innovation of the Transformer architecture, specifically self-attention.

The key innovation of the Transformer is the self-attention mechanism. It allows each token in a sequence to dynamically weigh the importance of all other tokens in the same sequence when computing its representation. This enables the model to capture long-range dependencies and contextual relationships without relying on recurrent connections, making it highly parallelizable and efficient.

6. Why is tokenization a necessary step before training a language model?

Tokenization converts raw text into a numerical sequence that a neural network can process. Language models operate on discrete units (tokens) rather than raw characters or words directly. Tokenization, especially subword methods like BPE, manages vocabulary size, handles out-of-vocabulary words, and creates meaningful input representations for the model to learn from.

Chapter 2: Code, Notes and Prerequisites

1. Why is a 'code-first' approach emphasized in this course, and what are two specific ways to implement it in your study routine?

A code-first approach is emphasized because it promotes deep understanding through active engagement, rather than passive observation. Two ways to implement this are by typing out all code yourself (avoiding copy-paste) and by intentionally 'breaking things' to predict and observe the effects of changes on the network's behavior.

2. What is the primary purpose of the `micrograd` repository, and how does it relate to the mathematical foundations reviewed in this chapter?

The `micrograd` repository is for building a tiny auto-differentiation engine from scratch. It directly relates to the mathematical foundations by demonstrating how derivatives and the chain rule are applied computationally to automatically calculate gradients, which is the core mechanism of backpropagation.

3. Explain how the chain rule is fundamental to the training process of neural networks.

The chain rule is fundamental because neural networks are composed of multiple layers, each representing a function. Backpropagation, the algorithm used to train neural networks, repeatedly applies the chain rule to compute the gradient of the loss function with respect to every parameter in the network, propagating these gradients efficiently from the output layer back to the input layer.

4. How are vectors and matrices used to represent data and operations within a neural network?

Vectors and matrices are used to represent input data, network parameters (weights and biases), and intermediate activations. Linear transformations within layers, such as multiplying inputs by weights and adding biases, are efficiently performed using matrix multiplication and vector addition, making linear algebra the operational language of neural networks.

5. Describe the roles of the softmax function and cross-entropy loss in a neural network designed for classification.

In a classification network, the softmax function takes the raw outputs (logits) and converts them into a probability distribution over the classes. Cross-entropy loss then measures the difference between this predicted probability distribution and the true class distribution, providing a quantifiable error signal that the network aims to minimize during training.

Chapter 3: Final Assessment: Neural Nets

1. What is the primary purpose of a loss function in a neural network?

The primary purpose of a loss function is to quantify the error or discrepancy between the neural network's predicted output and the actual target output. It provides a numerical value that indicates how well the model is performing. This value is then used by optimization algorithms to adjust the network's parameters.

2. Briefly explain the role of backpropagation in training a neural network.

Backpropagation is an algorithm that efficiently calculates the gradients of the loss function with respect to every weight and bias in the network. It does this by applying the chain rule of calculus backward through the network layers. These gradients are then used by an optimizer, like gradient descent, to update the network's parameters, thereby minimizing the loss.

3. How does the `micrograd` library help in understanding neural network training?

The `micrograd` library helps by providing a hands-on implementation of auto-differentiation from first principles. It allows students to see how gradients are computed for each individual operation and how they propagate backward through a computational graph. This demystifies the backpropagation process, showing its mathematical underpinnings in code.

4. What is the significance of the self-attention mechanism in a GPT-style language model?

The self-attention mechanism allows a GPT-style language model to weigh the importance of different words in an input sequence when processing each word. It enables the model to capture long-range dependencies and contextual relationships within the text. This mechanism is crucial for understanding sentence structure and generating coherent, contextually relevant text.

5. Describe one key difference between a simple multi-layer perceptron and a Transformer architecture.

One key difference is how they process sequential data. A simple multi-layer perceptron processes inputs independently or in fixed-size windows, lacking an inherent mechanism to understand long-range dependencies in sequences. A Transformer architecture, however, uses self-attention mechanisms to process entire sequences at once, allowing each element to attend to all other elements and capture global contextual relationships.

CLOSING

Where to Go Next

Completing "Neural Networks: Zero to Hero" equips you with foundational skills in building complex AI models. To consolidate your understanding, consider modifying the GPT-style model you built. Experiment with different architectures, activation functions, or datasets.

Practice implementing micrograd for new computational graphs or simpler neural network designs. Building variations of the course projects from scratch will deepen your practical knowledge.

From here, you can explore more advanced topics in deep learning, such as specific transformer architectures, reinforcement learning, or generative models beyond basic language models. Applying these concepts to real-world problems will further your expertise.



MEGACOURSE

Learn Anything. Achieve Everything.

Track your progress, take the quizzes, and earn your certificate for this course at
megacourse.app/courses/neural-networks-zero-to-hero